CROSSTALK

December 2007 The Journal of Defense Software Engineering Vol. 20 No. 12

Software Sustainment

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 20, Number 12, December 2007	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Software Sustainment

Software Engineering Technology

ON THE COVER
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

Departments

The CrossTalk staff would like to wish you and yours the very best this holiday season and the happiest of New Years.

CROSSTALK

CO-SPONSORS:
DoD-CIO *The Honorable John Grimes*
NAVAIR *Jeff Schwalb*
76 SMXG *Kevin Stamey*
309 SMXG *Norman LeClair*
402 SMXG *Diane Suchan*
DHS *Joe Jarzombek*

STAFF:
MANAGING DIRECTOR *Brent Baxter*
PUBLISHER *Elizabeth Starrett*
MANAGING EDITOR *Kase Johnstun*
ASSOCIATE EDITOR *Chelene Fortier-Lozancich*
ARTICLE COORDINATOR *Nicole Kentta*
PHONE (801) 775-5555
E-MAIL crosstalk.staff@hill.af.mil
CROSSTALK ONLINE www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); U.S. Navy (USN); U.S. Air Force (USAF); Defense Finance and Accounting Services (DFAS); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); Ogden-ALC 309 SMXG; and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 28.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Software Sustainment or Maintenance?

I work with a person who cringes when he hears the term *software maintenance*. For him, maintenance brings forth images of an angry general asking why he is paying to fix a system that he already paid for and expects to work. As you will read in this month's issue, software maintenance usually involves so much more and perhaps the term *software sustainment* is more descriptive.

As the CROSSTALK staff prepared this month's selection of articles, I noticed one software sustainment topic that warrants additional discussion: knowledge retention. In order to sustain software using the techniques discussed in this month's issue, knowledge of the software/system is needed. This required knowledge must be adequately planned in addition to other resources required for adequate logistics support. Basically, someone has to know how the software works. Even if the number of fixes, enhancements, alterations, and other activities are minimal and just a few engineers could conceivably address these, the size and complexity of the system may be quite large, requiring more people to cover the needed knowledge.

I am fortunate to work closely with Dr. Randall Jensen, one of the leaders in the software estimation community. In a recent discussion, he pointed to heuristics from Dr. Barry Boehm in his book, "Software Engineering Economics." These heuristics assign complexity values to various software systems such as operating systems, accounting systems, operational flight programs (OFP), etc. For example, an OFP is assigned a value of 10. This 10 equates to (of all things) boxes of cards that an operator can handle for this system. One can imagine this information is quite old because computer cards are certainly before my time. Yet the statistic seems to have stood the test of time. By allowing 2,000 cards per box, or 2,000 source lines of code (KSLOC) per number, a typical OFP will need a knowledgeable person for every 20 KSLOC (2 KSLOC x 10).

Of course, it is cost prohibitive to have numerous people waiting around to make few alterations to the code. However, these people can spend their excess time supporting other systems that may have oversight from other experts.

There are numerous other sustainment considerations within this month's CROSSTALK, beginning with Capers Jones' *Geriatric Issues of Aging Software*. In his article, Jones provides a much more comprehensive discussion of sustainment issues that must be considered when planning a logistics effort. We next share the F-35 Lightning II sustainment approach, including the contracting structure, in *Performance-Based Software Sustainment for the F-35 Lightning II* by Lloyd Huff and George Novak. In our final theme article, *Reference Metrics for Service-Oriented Architectures*, Dr. Yun-Tung Lau suggests including service time, scalability, availability, and reliability while measuring the usefulness of a fielded service-oriented architecture.

We offer two supporting articles this month, beginning with *A Primer on Java Obfuscation* by Stephen Torri, Derek Sanders, Gordon Evans, and Dr. Drew Hamilton. In this article, the authors caution on the attempted use of obfuscation to protect Java code, while adding some suggestions for consideration if Java is truly required in a secure system. Finally, Alison A. Frost and Michael J. Campo discuss defect containment in *Advancing Defect Containment to Quantitative Defect Management*.

Having worked on multiple sustainment efforts, I have experienced the daunting task of understanding the code that is being altered. What would have often been a simple job for a system I was familiar with, became more complicated and time-consuming as I needed to learn about the software before starting any changes. When this need is added to complex changes requiring requirements modeling, contracting, etc., the list of considerations is extensive and must be approached with educated knowledge.

Elizabeth Starrett
Publisher

Geriatric Issues of Aging Software

Capers Jones
Software Productivity Research, LLC.

Software has been a mainstay of business and government operations for more than 50 years. As a result, all large enterprises utilize aging software in significant amounts. Some companies exceed 5,000,000 function points in the total volume of their corporate software portfolios. Much of this software is now more than 10 years old, and some applications are more than 25 years old. Maintenance of aging software tends to become more difficult year by year since updates gradually destroy the original structure of the applications and increase its entropy. Aging software may also contain troublesome regions with very high error densities called error-prone modules. Repairs to aging software suffer from a phenomenon called bad fix injection, or new defects are accidentally introduced as a byproduct of fixing previous defects.

As the 21st century advances, more than 50 percent of the global software population is engaged in modifying existing applications rather than writing new applications. This fact by itself should not be a surprise because whenever an industry has more than 50 years of product experience, the personnel who repair existing products tend to outnumber the personnel who build new products. For example, there are more automobile mechanics in the United States who repair automobiles than there are personnel employed in building new automobiles.

The imbalance between software development and maintenance is opening up new business opportunities for software outsourcing groups. It is also generating a significant burst of research into tools and methods for improving software maintenance performance.

What Is Software Maintenance?

The word *maintenance* is surprisingly ambiguous in a software context. In normal usage, it can span some 23 forms of modification to existing applications. The two most common meanings of the word maintenance include the following: 1) defect repairs, and 2) enhancements (or adding new features to existing software applications).

Although software enhancements and software maintenance in the sense of defect repairs are usually funded in different ways and have quite different sets of activity patterns associated with them, many companies lump these disparate software activities together for budgets and cost estimates.

The author does not recommend the practice of aggregating defect repairs and enhancements, but this practice is very common. Consider some of the basic differences between enhancements or adding new features to applications and maintenance or defect repairs as shown in Table 1.

Because the general topic of *maintenance* is so complicated and includes so many different kinds of work, some companies merely lump all forms of maintenance together, using gross metrics such as the overall percentage of annual software budgets devoted to all forms of maintenance summed together. This method is crude, but can convey useful information. An organization that is proactive in using geriatric tools and services can spend less than 30 percent of its annual software budget on various forms of maintenance, while an organization that has not used any of the geriatric tools and services can top 60 percent of its annual budget on various forms of maintenance.

The kinds of maintenance tools used by lagging, average, and leading organizations are shown in Table 2. Table 2 is part of a larger study that examined many different kinds of software engineering and project management tools [1].

It is interesting that the leading companies in terms of maintenance sophistication not only use more tools than the laggards, but they use more of their features as well. Again, the function point values in Table 2 refer to the capabilities of the tools that are used in day-to-day maintenance operations. The leaders not only use more tools, but they do more with them.

Before proceeding, let us consider 23 discrete topics that are often coupled together under the generic term *maintenance* in day-to-day discussions, but which are actually quite different in many important respects [2] (See Table 3 for the list of 23 topics).

Although the 23 maintenance topics are different in many respects, they all have one common feature that makes a group discussion possible: They all involve modifying an existing application rather than starting from scratch with a new application.

Each of the 23 forms of modifying existing applications has a different reasons for being carried out. However, it often happens that several of them take place concurrently. For example, enhancements and defect repairs are very common in the same release of an evolving application. There are also common sequences or patterns to these modification activities. For example, reverse engineering often precedes reengineering and the two occur so often together as to almost comprise a linked set. For releases of large applications and major systems, the author has observed between six and 10 forms of maintenance all leading up to the same release.

Table 1: *Key Differences Between Maintenance and Enhancements*

	Enhancements (New features)	Maintenance (Defect repairs)
Funding source	Clients	Absorbed
Requirements	Formal	None
Specifications	Formal	None
Inspections	Formal	None
User documentation	Formal	None
New function testing	Formal	None
Regression testing	Formal	Minimal

Geriatric Problems of Aging Software

Once software is put into production it continues to change in three important ways:

1. Latent defects still present at release must be found and fixed after deployment.
2. Applications continue to grow and add new features at a rate of between 5 percent and 10 percent per calendar year, due either to changes in business needs or to new laws and regulations, or both.
3. The combination of defect repairs and enhancements tends to gradually degrade the structure and increase the complexity of the application. The term for this increase in complexity over time is called *entropy*. The average rate at which software entropy increases is about 1 percent to 3 percent per calendar year.

Because software defect removal and quality control are imperfect, there will always be bugs or defects to repair in delivered software applications. The current U.S. average for defect removal efficiency is only about 85 percent of the bugs or defects introduced during development [3] and has stayed almost the same for more than 10 years. The actual values are about five bugs per function point created during development. If 85 percent of these are found before release, about 0.75 bugs per function point will be released to customers. For a typical application of 1,000 function points or 100,000 source code statements, that implies about 750 defects present at delivery. About one-third – or 250 defects – will be serious enough to stop the application from running or create erroneous outputs.

Since defect potentials tend to rise with the overall size of the application, and since defect removal efficiency levels tend to decline with the overall size of the application, the overall volume of latent defects delivered with the application rises with size. This explains why super-large applications in the range of 100,000 function points, such as Microsoft Windows and many enterprise resource planning (ERP) applications, may require years to reach a point of relative stability. These large systems are delivered with thousands of latent bugs or defects.

Not only is software deployed with a significant volume of latent defects, but a phenomenon called *bad fix injection* has been observed for more than 50 years. Roughly 7 percent of all defect repairs will contain a new defect that was not there before. For very complex and poorly structured applications, these bad-fix injections have topped 20 percent [3].

Maintenance Engineering	Lagging	Average	Leading
Reverse engineering		1,000	3,000
Reengineering		1,250	3,000
Code restructuring			1,500
Configuration control	500	1,000	2,000
Test support		500	1,500
Customer support		750	1,250
Debugging tools	750	750	1,250
Defect tracking	500	750	1,000
Complexity analysis			1,000
Mass update search engines		500	1,000
Function point subtotal	*1,750*	*6,500*	*16,500*
Number of tools	*3*	*8*	*10*

Table 2: *Numbers and Size Ranges of Maintenance Engineering Tools (Size data expressed in terms of function point metrics)*

In the 1970s, IBM did a distribution analysis of customer-reported defects against their main commercial software applications. The IBM personnel involved in the study, including the author, were surprised to find that defects were not randomly distributed through all of the modules of large applications [4].

In the case of IBM's main operating system, about 5 percent of the modules contained just over 50 percent of all reported defects. The most extreme example was a large database application, where 31 modules out of 425 contained more than 60 percent of all customer-reported bugs. These troublesome areas were known as *error-prone modules*.

Similar studies by other corporations such as AT&T and ITT found that error-prone modules were endemic in the software domain. More than 90 percent of applications larger than 5,000 function points were found to contain error-prone modules in the 1980s and early 1990s. Summaries of the error-prone module data from a number of companies was published in [3].

Fortunately, it is possible to surgically remove error-prone modules once they are identified. It is also possible to prevent them from occurring. A combination of defect measurements, formal design inspections, formal code inspections, and formal testing and test-coverage analysis have proven to be effective in preventing error-prone modules from coming into existence [5].

Today in 2007, error-prone modules are almost nonexistent in organizations

Table 3: *Major Kinds of Work Performed Under the Generic Term Maintenance*

Major Kinds of Work Performed Under the Generic Term *Maintenance*
1. Major enhancements (new features of > 20 function points).
2. Minor enhancements (new features of < 5 function points).
3. Maintenance (repairing defects for good will).
4. Warranty repairs (repairing defects under formal contract).
5. Customer support (responding to client phone calls or problem reports).
6. Error-prone module removal (eliminating very troublesome code segments).
7. Mandatory changes (required or statutory changes).
8. Complexity or structural analysis (charting control flow plus complexity metrics).
9. Code restructuring (reducing cyclomatic and essential complexity).
10. Optimization (increasing performance or throughput).
11. Migration (moving software from one platform to another).
12. Conversion (changing the interface or file structure).
13. Reverse engineering (extracting latent design information from code).
14. Reengineering (transforming legacy application to modern forms).
15. Dead code removal (removing segments no longer utilized).
16. Dormant application elimination (archiving unused software).
17. Nationalization (modifying software for international use).
18. Mass updates such as the Euro or Year 2000 (Y2K) repairs.
19. Refactoring, or reprogramming, applications to improve clarity.
20. Retirement (withdrawing an application from active service).
21. Field service (sending maintenance members to client locations).
22. Reporting bugs or defects to software vendors.
23. Installing updates received from software vendors.

that are higher than Level 3 on the Software Engineering Institute's Capability Maturity Model® (CMM®). However, they remain common and troublesome for Level 1 organizations and for organizations that lack sophisticated quality measurements and quality control.

If the author's clients are representative of the United States as a whole, more than 50 percent of U.S. companies still do not utilize the CMM at all. Of those who do use the CMM, less than 15 percent are at Level 3 or higher. That implies that error-prone modules may exist in more than half of all large corporations and in a majority of state government software applications as well.

Once deployed, most software applications continue to grow at annual rates of between 5 percent and 10 percent of their original functionality. Some applications, such as Microsoft Windows, have increased in size by several hundred percent over a 10-year period.

The combination of continuous growth of new features coupled with continuous defect repairs tends to drive up the complexity levels of aging software applications. Structural complexity can be measured via metrics such as cyclomatic and essential complexity using a number of commercial tools. If complexity is measured on an annual basis and there is no deliberate attempt to keep complexity low, the rate of increase is between 1 percent and 3 percent per calendar year.

However – and this is an important fact – the rate at which entropy or complexity increases is directly proportional to the initial complexity of the application. For example, if an application is released with an average cyclomatic complexity level of less than 10, it will tend to stay well structured for at least five years of normal maintenance and enhancement changes.

But if an application is released with an average cyclomatic complexity level of more than 20, its structure will degrade rapidly and its complexity levels might increase by more than 2 percent per year. The rate of entropy and complexity will even accelerate after a few years.

As it happens, both bad-fix injections and error-prone modules tend to correlate strongly (although not perfectly) with high levels of complexity. A majority of error-prone modules have cyclomatic complexity levels of 10 or higher. Bad-fix injection levels for modifying high-complexity applications are often higher than 10 percent.

In the late 1990s, a special kind of geriatric issue occurred which involved making simultaneous changes to thousands of software applications. The first of these *mass update* geriatric issues was the deployment of the Euro currency, which required changes to currency conversion routines in thousands of applications. The Euro was followed almost immediately by the dreaded Y2K (Year 2000) problem [6], which also involved mass updates of thousands of applications. More recently in March of 2007, another such issue occurred when the starting date of daylight savings time was changed.

Future mass updates will occur later in the century when it may be necessary to add another digit to telephone numbers or area codes. Yet another and very serious mass update will occur if it becomes necessary to add digits to social security numbers in the second half of the 21st century. There is also the potential problem of the Unix time clock expiration in 2038.

Table 4: *Impact of Key Adjustment Factors on Maintenance (sorted in order of maximum positive impact)*

Maintenance Factors	Plus Range
Maintenance specialists	35%
High staff experience	34%
Table-driven variables and data	33%
Low complexity of base code	32%
Test coverage tools and analysis	30%
Code restructuring tools	29%
Reengineering tools	27%
High-level programming languages	25%
Reverse engineering tools	23%
Complexity analysis tools	20%
Defect tracking tools	20%
Mass update specialists	20%
Automated change control tools	18%
Unpaid overtime	18%
Quality measurements	16%
Formal base code inspections	15%
Regression test libraries	15%
Excellent response time	12%
Annual training of > 10 days	12%
High management experience	12%
Help-desk automation	12%
No error prone modules	10%
Online defect reporting	10%
Productivity measurements	8%
Excellent ease of use	7%
User satisfaction measurements	5%
High team morale	5%
Sum	*503%*

Metrics Problems With Small Maintenance Projects

There are several difficulties in exploring software maintenance costs with accuracy. One of these difficulties is the fact that maintenance tasks are often assigned to development personnel who interweave both development and maintenance as the need arises. This practice makes it difficult to distinguish maintenance costs from development costs because the programmers are often rather careless in recording how time is spent.

Another and very significant problem is the fact that a great deal of software maintenance consists of making very small changes to software applications. Quite a few bug repairs may involve fixing only a single line of code. Adding minor new features, such as a new line-item on a screen, may require less than 50 source code statements.

These small changes are below the effective lower limit for counting function point metrics. The function point metric includes weighting factors for complexity, and even if the complexity adjustments are set to the lowest possible point on the scale, it is still difficult to count function points below a level of perhaps 15 function points [7].

Quite a few maintenance tasks involve changes that are either a fraction of a function point, or may at most be less than 10 function points or about 1,000 COBOL source code statements. Although normal counting of function points is not feasible for small updates, it is possible to use the *backfiring* method or converting counts of logical source code statements into equivalent function points. For example, suppose an update requires adding 100 COBOL statements to an existing application. Since it usually takes about 105 COBOL statements in the procedure and data divisions to encode one function point, it can be stated that this small maintenance project is *about one function point in size*.

If the project takes one work day consisting of six hours, then at least the results can be expressed using common metrics. In this case, the results would be roughly six staff hours per function point. If the reciprocal metric *function points per staff month* is used, and there are 20 working days in the month, then the results would be 20 *function points per staff month*.

Best and Worst Practices in Software Maintenance

Because maintenance of aging legacy software is labor intensive, it is quite important to explore the best and most cost effective methods available for dealing with the millions of applications that cur-

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

rently exist. The sets of best and worst practices are not symmetrical. For example, the practice that has the most positive impact on maintenance productivity is the use of trained maintenance experts. However, the factor that has the greatest negative impact is the presence of *error-prone modules* in the application that is being maintained.

Table 4 illustrates a number of factors which have been found to exert a beneficial positive impact on the work of updating aging applications and shows the percentage of improvement compared to average results.

At the top of the list of maintenance *best practices* is the utilization of full-time, trained maintenance specialists rather than turning over maintenance tasks to the untrained generalists. Trained maintenance specialists are found most often in two kinds of companies: 1) large systems software producers such as IBM, and 2) large maintenance outsource vendors. The curricula for training maintenance personnel can include more than a dozen topics and the training periods range from two weeks to a maximum of about four weeks.

Since training of maintenance specialists is the top factor, Table 5 shows a modern maintenance curriculum such as those found in large maintenance outsource companies.

The positive impact from utilizing maintenance specialists is one of the reasons why maintenance outsourcing has been growing so rapidly. The maintenance productivity rates of some of the better maintenance outsource companies is roughly twice that of their clients prior to the completion of the outsource agreement. Thus, even if the outsource vendor costs are somewhat higher, there can still be useful economic gains.

Let us now consider some of the factors that exert a negative impact on the work of updating or modifying existing software applications. Note that the top-ranked factor that reduces maintenance productivity, the presence of error-prone modules, is very asymmetrical. The absence of error-prone modules does not speed up maintenance work, but their presence definitely slows down maintenance work.

In general, more than 80 percent of latent bugs found by users in software applications are reported against less than 20 percent of the modules. Once these modules are identified then they can be inspected, analyzed, and restructured to reduce their error content down to safe levels.

Software Maintenance Courses	Days	Sequence
Error-Prone Module Removal	2.00	1
Complexity Analysis and Reduction	1.00	2
Reducing Bad Fix Injections	1.00	3
Defect Reporting and Analysis	0.50	4
Change Control	1.00	5
Configuration Control	1.00	6
Software Maintenance Workflows	1.00	7
Mass Updates to Multiple Applications	1.00	8
Maintenance of Commercial Off-The-Shelf Packages	1.00	9
Maintenance of ERP Applications	1.00	10
Regression Testing	2.00	11
Test Library Control	2.00	12
Test Case Conflicts and Errors	2.00	13
Dead Code Isolation	1.00	14
Function Points for Maintenance	0.50	15
Reverse Engineering	1.00	16
Reengineering	1.00	17
Refactoring	0.50	18
Maintenance of Reusable Code	1.00	19
Object-Oriented Maintenance	1.00	20
Maintenance of Agile and Extreme Code	1.00	21
TOTAL	*23.50*	

Table 5: *Sample Maintenance Curricula for Companies Using Maintenance Specialists*

Table 6 summarizes the major factors that degrade software maintenance performance. Not only are error-prone modules troublesome, but many other factors can degrade performance too. For example, very complex *spaghetti code* is quite difficult to maintain safely. It is also troublesome to have maintenance tasks assigned to generalists rather than to trained maintenance specialists.

A common situation that often degrades performance is lack of suitable maintenance tools, such as defect tracking software, change management software, test library software, and so forth. In general, it is easy to botch-up maintenance and make it such a labor-intensive activity that few resources are left over for development work.

The last factor in Table 6, no unpaid overtime, deserves a comment. Unpaid overtime is common among software maintenance and development personnel. In some companies it amounts to about 15 percent of the total work time. Because it is unpaid it is usually unmeasured. That means side-by-side comparisons of productivity rates or costs between groups with unpaid overtime and groups without will favor the group with unpaid overtime because so much of their work is uncompensated and, hence, invisible. This is a benchmarking trap for

Maintenance Factors	Minus Range
Error-prone modules	-50%
Embedded variables and data	-45%
Staff inexperience	-40%
High complexity of base code	-30%
Lack of test coverage analysis	-28%
Manual change control methods	-27%
Low-level programming languages	-25%
No defect tracking tools	-24%
No *mass update* specialists	-22%
Poor ease of use	-18%
No quality measurements	-18%
No maintenance specialists	-18%
Poor response time	-16%
Management inexperience	-15%
No base code inspections	-15%
No regression test libraries	-15%
No help-desk automation	-15%
No on-line defect reporting	-12%
No annual training	-10%
No code restructuring tools	-10%
No reengineering tools	-10%
No reverse engineering tools	-10%
No complexity analysis tools	-10%
No productivity measurements	-7%
Poor team morale	-6%
No user satisfaction measurements	-4%
No unpaid overtime	0%
Sum	*-500%*

Table 6: I*mpact of Key Adjustment Factors on Maintenance (sorted in order of maximum negative impact)*

the unwary. Because excessive overtime is psychologically harmful if continued over long periods, it is unfortunate that unpaid overtime tends to be ignored when benchmark studies are performed.

Given the enormous amount of effort that is now being applied to software maintenance, and which will be applied in the future, it is obvious that every corporation should attempt to adopt maintenance *best practices* and avoid maintenance *worst practices* as rapidly as possible.

Software Entropy and Total Cost of Ownership

The word *entropy* means the tendency of systems to destabilize and become more chaotic over time. Entropy is a term from physics and is not a software-related word. However, entropy is true of all complex systems, including software. All known compound objects decay and become more complex with the passage of time unless effort is exerted to keep them repaired and updated. Software is no exception. The accumulation of small updates over time tends to gradually degrade the initial structure of applications and makes changes grow more difficult over time.

For software applications, entropy has long been a fact of life. If applications are developed with marginal initial quality control they will probably be poorly structured and contain error-prone modules. This means that every year, the accumulation of defect repairs and maintenance updates will degrade the original structure and make each change slightly more difficult. Over time, the application will destabilize and *bad fixes* will increase in number and severity. Unless the application is restructured or fully refurbished, it eventually will become so complex that maintenance can only be performed by a few experts who are more or less locked into the application.

By contrast, leading applications that are well structured initially can delay the onset of entropy. Indeed, well-structured applications can achieve declining maintenance costs over time. This is because updates do not degrade the original structure, as happens in the case of *spaghetti bowl* applications where the structure is almost unintelligible when maintenance begins.

The total cost of ownership of a software application is the sum of six major expense elements: 1) the initial cost of building an application, 2) the cost of enhancing the application with new features over its lifetime, 3) the cost of repairing defects and bugs over the application's lifetime, 4) the cost of customer support for fielding and responding to queries and customer-reported defects, 5) the cost of periodic restructuring or *refactoring* of aging applications to reduce entropy and thereby reduce bad-fix injection rates, and 6) removal of error-prone modules via surgical removal and redevelopment. This last expense element will only occur for legacy applications that contain error-prone modules.

Similar phenomena can be observed outside of software. Hypothetically, if you buy an automobile that has a high frequency of repair as shown in Consumer Reports and you skimp on lubrication and routine maintenance, you will fairly soon face some major repair problems – usually well before 50,000 miles. By contrast, if you buy an automobile with a low frequency of repair as shown in Consumer Reports and you are scrupulous in maintenance, you should be able to drive the car more than 100,000 miles without major repair problems.

Summary and Conclusions

In every industry, maintenance tends to require more personnel than building new products. For the software industry, the number of personnel required to perform maintenance is unusually large and may soon top 70 percent of all technical software workers. The main reasons for the high maintenance efforts in the software industry are the intrinsic difficulties of working with aging software. Special factors such as *mass updates* that began with the roll-out of the Euro and the Y2K problem are also geriatric issues.

Given the enormous efforts and costs devoted to software maintenance, every company should evaluate and consider best practices for maintenance and should avoid worst practices if at all possible.◆

References

1. Jones, Capers. "Analyzing the Tools of Software Engineering." Software Productivity Research (SPR) Technical Report. Burlington, MA: 1999.
2. Jones, Capers. Estimating Software Costs. 2nd ed. McGraw Hill, 1998.
3. Jones, Capers. Software Quality – Analysis and Guidelines for Success. Boston, MA: International Thomson Computer Press, 1997.
4. Jones, Capers. "Program Quality and Programmer Productivity." IBM Technical Report. TR 02.764. San Jose, CA: IBM, 1977.
5. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Boston, MA: Addison Wesley Longman, 2000.
6. Jones, Capers. The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences. Reading, MA: Addison Wesley, 1998.
7. Jones, Capers. Applied Software Measurement. 2nd ed. McGraw Hill, 1996.

Additional Reading

1. Arnold, Robert S. Software Reengineering. IEEE. Los Alamitos, CA: Computer Society Press, 1993.
2. Arthur, Lowell Jay. Software Evolution – The Software Maintenance Challenge. New York: John Wiley & Sons, 1988.
3. Gallagher, R.S. Effective Customer Support. Boston, MA: International Thomspon Computer Press, 1997.
4. Kan, Stephen H. Metrics and Models in Software Quality Engineering. Reading, MA: Addison Wesley, 2003.

About the Author

Capers Jones is currently the chairman of Capers Jones and Associates, LLC. He is also the founder and former chairman of SPR, where he holds the title of Chief Scientist Emeritus. He is a well-known author and international public speaker, and has authored the books "Patterns of Software Systems Failure and Success," "Applied Software Measurement," "Software Quality: Analysis and Guidelines for Success," "Software Cost Estimation," and "Software Assessments, Benchmarks, and Best Practices." Jones and his colleagues from SPR have collected historical data from more than 600 corporations and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods. The total volume of projects studied now exceeds 12,000.

Software Productivity Research, LLC
Phone: (877) 570-5459
Fax: (781) 273-5176
E-mail: capers.jones@spr.com, info@spr.com

Performance-Based Software Sustainment for the F-35 Lightning II

Lloyd Huff and George Novak
Lockheed Martin Aeronautics

The complexity and sophistication of F-35 Air System software and the multiplicity of F-35 missions, versions, and customers, combined with a performance-based contract structure, present unprecedented software sustainment challenges. Understanding how F-35 software will be sustained is the focus of ongoing analysis and planning. This article describes some of the revolutionary conclusions and products of that analysis and provides a look forward to performance-based sustainment of software for the multinational F-35 fleet.

In mid-2007, an article entitled "Lockheed Martin Hopes F-35 Leads to Maintenance Revolution" [1] was released through internet news outlets. The article stated the following:

> Lockheed Martin ... is designing a new kind of maintenance program for the $300 billion F-35 Joint Strike Fighter project, which company officials say could set a new standard for military aircraft operations. The U.S.-led, nine-nation fighter program has "performance-based logistics" built into its purchase plan, giving contractors a big role in maintenance management. ... Lockheed Martin says this maintenance strategy means that logistical support will make up about 50 percent of total program costs, compared to 67 percent of total costs under a less centralized strategy ... Under the maintenance plan, F-35 owners will pay for anticipated operating time. ... Based on how much the aircraft are expected to fly, Lockheed Martin will manage parts inventory, plan overhaul schedules and train the military crews who support aircraft operations. ... The F-35 program marks the first time an entire aircraft has used a performance-based logistics plan. Such pay-by-the-flight-hour maintenance strategies are more common for component systems, or commercial jet operations.

The ramifications of this *maintenance revolution* to software engineering and management are extensive. Initial planning for sustainment of F-35 software began in 2002. In 2005, however, the Office of the Secretary of Defense confirmed that a performance-based sustainment approach would be applied to the F-35 Joint Strike Fighter (JSF) program. This decision focused the planning phase and allowed more detailed analysis to begin. The results of the analysis to date are reviewed in this article. These results are shaping the approach to F-35 software sustainment in order to support air system performance and life-cycle cost savings objectives.

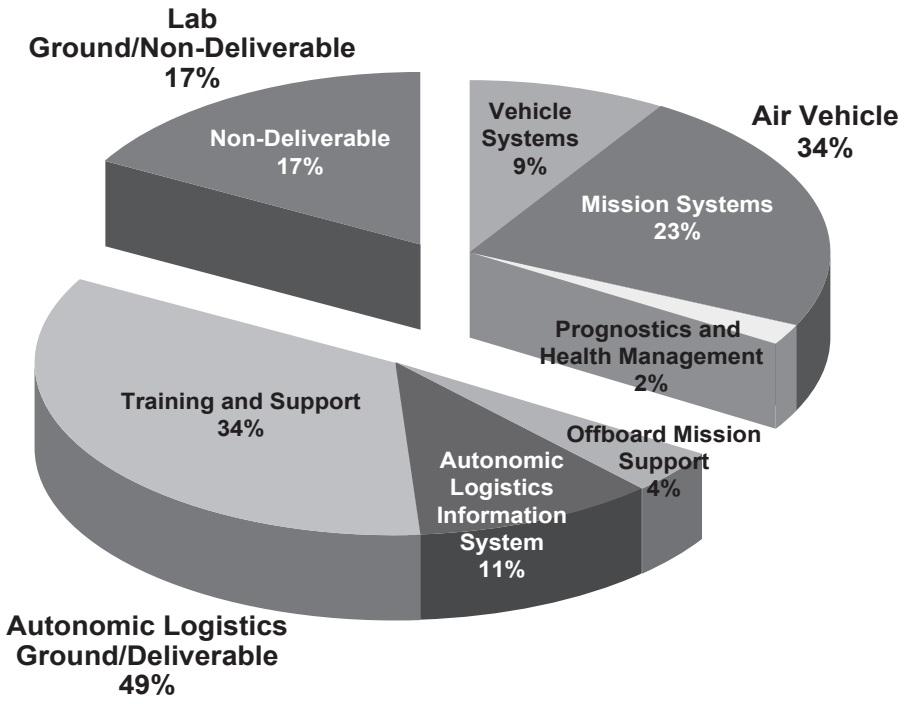
F-35 Program Overview

The F-35 Air System consists of the Air Vehicle (AV), including the propulsion system and the Autonomic Logistics (AL) system. The AV is a three-variant family of fifth-generation[1] strikefighter aircraft consisting of the F-35A Conventional Take-off and Landing (CTOL), F-35B Short Take-off Vertical Landing (STOVL), and F-35C Carrier Variant (CV).

A high degree of designed-in commonality will exist among the three variants (e.g., engines, avionics, crew station, subsystems, suspension, and release equipment and structure). CTOL operations will be a common capability among the variants with unique capabilities for the CV (e.g., catapult and arresting gear compatibility) and STOVL (e.g., vertical launch and recovery, and ski jump compatibility) variants. Key features include a blend of supportable low observable technologies, highly integrated mission systems, interchangeable propulsion systems, interoperability and internal and external carriage of stores. Other examples of commonality include Prognostics and Health Management systems, Institute of Electrical and Electronics Engineers (IEEE) 1394 aircraft bus design [2], and a cockpit incorporating advanced on-board and off-board sensor fusion. Each variant will provide an adverse weather, day/night capability to effectively execute operational missions.

The F-35 AV operates in concert with AL, including AL Information System (ALIS), which uses prognostics and health information from the AV to enable proactive maintenance. AL also features a training system which is concurrent with aircraft versions, missions, and maintenance tasks. And, F-35 AV and ground systems are designed to interoperate with the net-centric combat and logistics environments required for modern combat operations.

Figure 1: *Estimated F-35 Software Sustainment Baseline*



The air system software configuration present at the end of the system development and demonstration phase of the program forms the software sustainment baseline. This baseline, consisting of AV, AL systems, and lab software is currently estimated to be approximately 20 million source lines of code. A break-out by category is shown in Figure 1 (see previous page). Maintaining maximum commonality of this software across all variants and versions is key to achieving program affordability goals.

Figure 2 overlays the relationship of planned F-35 program phases (top of figure) to a standard product life cycle (center), and to the steps involved in development and delivery of product support (bottom of figure), through the end of program. The JSF Development and Low Rate Initial Production (LRIP) phases of the program ensure that the Air System and its support systems are mature as Full Rate Production (FRP) and Initial Operation Capability thresholds are reached. Software sustainment plans and estimates, for the *In-Service* portion of Figure 2, extend 50 years.

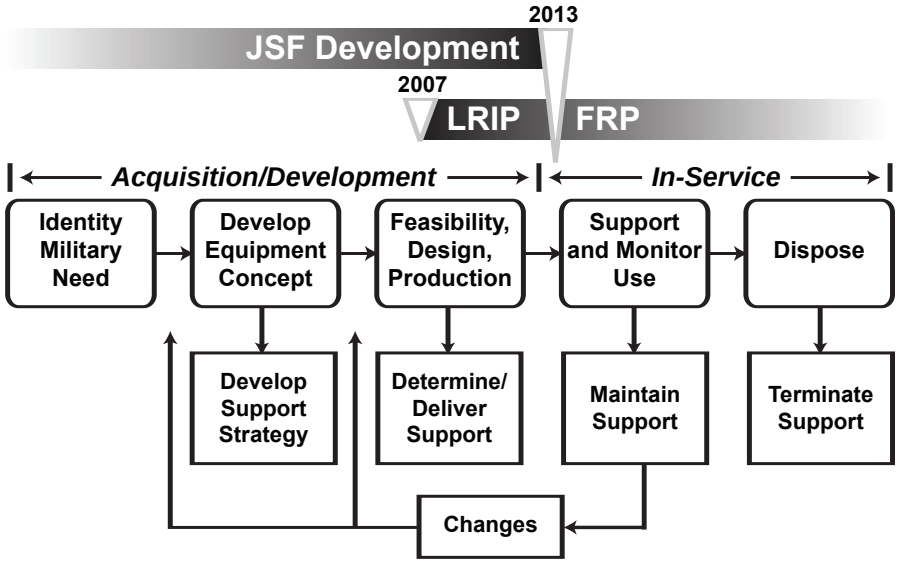
Performance-Based Contracting

The F-35 program includes partner participation by the U.S. Air Force, Navy, and Marine Corps; the United Kingdom; Italy; the Netherlands; Turkey; Canada; Australia; Denmark; and Norway. Additional foreign military sales are under consideration. Warfighters from these militaries will channel their needs through the JSF Program Office (JPO). A joint agreement on F-35 production, sustainment and follow-on development will guide the evolution of the Air System, and sets ground rules for partner participation.

The JPO is the single point of contractual direction from warfighters to JSF principal partners, and to propulsion system contractors. JSF principal partners include Lockheed Martin Aeronautics, (the Product Support Integrator [PSI]), Northrop Grumman, and BAE SYSTEMS. The PSI is, in turn, responsible for managing the F-35 global industrial base of United States and international suppliers and depots.

The performance-based contracting model can be visualized as a *loop*, which begins and ends with the warfighters and flows through the JPO, the PSI, and the global industrial base. Warfighters express their needed capabilities or changes to the JPO, along with their required performance levels expressed in terms of mission effectiveness, aircraft availability, sorties per month, etc. Performance-based contracts from JPO then transfer the risk and responsibility to provide specified performance levels to the PSI and to the industrial base. Metrics quantify air system performance and incentives or penalties. These performance metrics are the basis for a variable pricing component, referred to as *power by the hour*. Payment, under performance based contracting, is thus based on usage instead of breakage. Additionally, price improvement targets/ curves are established to drive reduction in cost over the term of a contract. In the event that design changes are implemented to improve performance, resulting cost reductions are shared between customers (in reduced price), and industry, (in increased profit). The point of performance-based contracts is encapsulated in the term Performance Based Outcomes (PBO). *Warfighters are now contracting for an outcome, or a result, as opposed to contracting for repairs, replacements, supplies, inventory, shipment, or services.*

Figure 2: *Mapping of Standard Life-Cycle Phases to the F-35 Program*



Managing Software in a Performance-Based Environment

Just as performance of the F-35 Air System is predicated on software, so is the success of performance-based contracting. Software is viewed as a crucial commodity among many that must be managed for predictability. This article will proceed to examine the keys to successful sustainment of software in a PBO environment. First, however, the PBO software sustainment domain will be scoped by reviewing boundaries, definitions, and success criteria.

As an entry point to analysis of software sustainment, a boundary graphic (Table 1), was produced to delineate software services funded under PBO, as opposed to that funded otherwise. To summarize the graphic, any software sustainment action taken to maintain the delivered software baseline falls *within* PBO; any software sustainment action which adds or changes functionality to the software baseline falls *outside* of PBO.

These boundaries, however, must be accompanied by precise definitions of software changes, releases, and support services. Software changes are defined in terms of *priority* (routine, urgent, and emergency) and *purpose* (corrective, adaptive, perfective, new capability, performance initiative, technology sustainment, or technology insertion). Software release types are defined in terms of size and *tempo* (such as major or minor block releases, AV maintenance updates, or asynchronous releases of AL software). Typical timespans for development are associated with software change and release categories. These values form the basis for sustainment cost models and a single, integrated master sustainment plan.

Measurement and analysis of F-35 software performance will be driven by performance-based metrics derived from Performance-Based Agreements between the JPO and the warfighters. The top-level metrics are decomposed into a metrics taxonomy. This metrics taxonomy encompasses all influences upon the top-level PBO metrics. Accordingly, software performance is measured and analyzed to quantify its operational performance. The lower-tier software metrics reveal the influence of software on F-35 air system PBO metrics and serve to initiate improvements in performance and predictability. Table 2 provides a look at how software might influence air system performance and how such influences will be tracked.

Keys to Successful PBO Sustainment of F-35 Software

As stated earlier, the ramifications of per-

formance-based contracting to F-35 software are far-reaching. Looking forward through 2063, factoring in the rate of technological change and considering security and safety ramifications, sustainment of F-35 software quickly moves from *far-reaching* to *prodigious*. As such, the following eight key steps are being taken to manage this commodity.

1. Strive for Commonality

The JSF program, from its inception, has been built upon the following four pillars: affordability, lethality, survivability, and supportability. The extent to which a common software baseline is retained across F-35 variants and F-35 international partners will directly affect overall affordability and supportability. While air system software is tailorable and compatible with each owning service's support environments, continued emphasis on commonality will maximize affordability and supportability for all system users. A common solution, employing minimal infrastructure, provides best value sustainment capability at minimum cost to all parties.

The ultimate goal of all participants, therefore, is to reach consensus on a common sustainment solution and, thereby, minimize the incidence of multiple system/software configurations. However, some unique capabilities will be necessary to satisfy specific operational needs, address sovereign requirements, and alleviate political and industrial concerns. Unique software capabilities will typically occur in two areas: in the AV Mission Systems software configuration (specifically, in weapons controls, pilot-vehicle interface, and communications/interoperability), or as additions to the F-35 software integration and test supporting infrastructure. In either event, F-35 partner countries will have the opportunity to have their changes considered for inclusion in the common baseline before steps are taken to assess the cost and impacts of a unique software change. Any unique functionality will be encapsulated to minimize re-verification expense and the cost will be borne by the partner/partners involved on a *pay-to-be-different* basis.

Covered by PBO Contract	Covered by Follow-On Development Contract
• Maintenance – Corrective Maintenance – Urgent and Emergency Corrections • Performance Initiatives – Operational Administration – Day-to-Day Administrative Support • Sustaining Engineering – Planning – Studies – Standing Boards, Configuration Management – Needs Analysis – Lab and Development – Environment/Infrastructure – Programming Infrastructure – Field Deficiency Assessment • Technology Sustainment	• New Capability • Technology Insertion • Adaptive Change • Perfective Change • Urgent Operational Need • Emergency Operational Need • Production Retrofit

Table 1: *Performance-Based Contracting Boundary*

2. Apply Industrial Engineering Practices to Software

Many parameters must be considered to plan and manage the performance-based F-35 software sustainment domain with a globally dispersed, multinational future fleet. These parameters include the frequency and quantity of software changes, the number of versions, and the time required for development, validation, and distribution. AV load times are important, even more so when viewed over a 50-year period. In this context, F-35 software sustainment emerges as an industrial engineering field, where efficiency, consistency, the elimina-

Table 2: *Software Influence on Performance-Based Metrics*

Performance Criteria	Top-Level PBO Metrics	Software Metrics
Readiness/ Availability	• Aircraft Availability • Mission Capable Aircraft Availability (AA) Rate	• Aircraft Downtime (Software) • Software Mission Capability
Mission Effectiveness	• Mission Effectiveness – Primary Task Not Accomplished; System Condition – Secondary Task Not Accomplished; System Condition	• Primary Task Not Accomplished (Software) • Secondary Task Not Accomplished (Software)
Required Sorties and Flying Hours Accomplished	• Percent Sorties Flown • Percent Flying Hours Flown (* Flown *Ground Abort *Cancelled)	• Percent Sorties Not Flown Due to Software • Percent Flying Hours Not Flown Due to Software
Logistics Footprint	• Logistics Footprint Data Total Change = Support Equipment Change + Personnel Change	• Support Equipment Change Due to Software – Support Equipment Size Change Due to Software – Support Equipment Quantity Change Due to Software • Personnel Change Due to Software – Direct Manpower Change Due to Software – Other Manpower Change Due to Software
Military Level of Effort	• Cannibalizations per 1,000 Flight Hours (FH) • Maintenance Man-Hours per FH • Maintenance Man-Hours per FH (A/C Subsystem)	• No Software Metric Applicable • Software Maintenance Man-Hours per Flight Hour • Software Maintenance Man-Hours per Flight Hour (A/C Subsystem)

tion of waste, and a solid understanding of capacity must be achieved. To help bring order and quantification to the F-35 software industry, modeling and measurement are being employed.

An end-to-end software sustainment process model supports the business case analysis of F-35 software sustainment. The model is expressed as an event-driven process model using the Architecture of Integrated Information Systems modeling tool (ARIS). ARIS was selected based on its features, its standardization within Lockheed Martin, and its use in development of interfacing models (F-35 AL operation guides, software integration and test lab modeling, system build, software loading processes, and software distribution). The software sustainment model starts with the field report of a software defect and ends with measurement of performance of the delivered, operational software solution. The model provides a means to understand tasks and capacity constraints, and supports estimation of sustainment costs.

An example analysis area within the model is the process of system build. System build is the packaging of hundreds of lower-tier software products to create a release product set. A complete air system build package is a single, deliverable software product (end item) to the fleet, for installation and operation. It is an organized assembly of vehicle system, mission system, and AL software, along with release documentation, flight clearance, technical data, and other supporting version information to facilitate identification and distribution. System build is a process that will be performed many times during each maintenance release cycle and is critical to both the delivery timeline and overall load integrity.

Tight controls are applied to this crucial *handover* point. The system build/final software integration process is dependent upon safeguards and controls imposed upon lower-tier software builds. All required certifications, qualifications, formats, and approvals must be applied throughout the software build hierarchy. Accordingly, three checkpoints/readiness *gates* were established to ensure that files and artifacts obtained for system build are completed, correctly formatted, fully described, and duly authorized. The gates affirm that all required software components, along with related files or data, are available and have been properly identified, are functionally acceptable, have achieved all required certifications and qualifications, and that interfaces comply with applicable requirements and descriptions. Transfer of files and artifacts through these gates are controlled with checklists, which specify criteria and are administered through a review and approval process. To the extent possible, additional safeguards have been incorporated in tools and workflows.

Apart from measuring the *integrity* of the system build process, a set of metrics is maintained to enable *capacity* planning for system build. As releases are produced, span times, touch times, execution times, delay times, and total effort metrics are tracked for routine and urgent builds. The results are synthesized into cost estimates and process improvement initiatives.

Table 3: *Software Maintenance Benchmarking Participants*

Benchmarking Projects	
British AV-8B	U.S. AV-8B
A-10	B-1B
B-2	C-17
C-130	JE-3
F-15	F-16
F-18	F-22
F-117	P-3C

3. Engage Customers

The F-35 software life cycle was planned in progressive stages. Each stage engaged users and partner country subject matter experts. First, U.S., U.K., and international standards for software maintenance and support, including IEEE, Society of Automotive Engineers, and military standards were canvassed to form a foundation for sustainment planning.

Next, fact-finding was accomplished through a benchmarking study of software maintenance operations across 14 military aircraft programs. A list of software maintenance operations willing to share their expertise is shown in Table 3.

A questionnaire was developed by F-35 Integrated Product Teams (IPTs). The benchmarking study manager used the questionnaire to conduct interviews with 39 representatives from the 14 software maintenance programs. The information obtained from these interviews was compiled and summarized in a report. The study served to identify practices which required consideration for adoption, or avoidance, by the F-35 program. It affirmed the importance of thorough planning, establishment of communication channels and information flows, and compliance with clearly documented processes. The study also revealed that successful support of multinational customers requires focused attention on several elements (e.g., export controls, software storage and segregation, and joint acceptance criteria for software changes).

Following the benchmarking study, results were incorporated into the software life-cycle plan for the F-35. The plan was subject to several rounds of review by graybeard panels. Each round of review was conducted during a 30-day timeframe, beginning with a kickoff, followed by individual preparation. Panelists invested an average of 8.5 hours to study the plan, rate the contents, and prepare their preliminary comments. Deep-dive assignments were allocated to focus specific panelists on selected topics. Panels were then convened for a face-to-face walk-through of topics over the course of several days, and results were summarized in outbriefs. More than 1,000 comments and recommendations were raised and addressed as the result of the graybeard panels. (Responses to several of these recommendations are noted throughout this article.)

Interactions with graybeard panelists opened the door for visits to seven U.S and international software maintenance operations by F-35 representatives. These visits resulted in useful dialogue with customers and software sustainment personnel from 11 aircraft programs. Again, a standard list of topics and questions were used to ensure consistency.

Finally, with a reasonably mature life-cycle plan in place, subject matter experts from partner countries were directly engaged with JSF contractors in a Software Maintenance and Sustainment Working Group (SMS WG), a team of 40 participants comprised of equal part contractors and customers. The SMS WG is chartered to ensure customer expectations relative to JSF software sustainment are considered in communicating, planning, documenting, contracting, and scheduling of affordable software sustainment solutions.

4. Adopt a Holistic Approach to Sustainment

Earlier in this article, the contractual boundaries between PBO maintenance changes and follow-on development changes were emphasized. While this is keenly important from a funding standpoint, PBO-driven software changes cannot be viewed independently. PBO-driven software changes must be weighed in the context of performance of the global F-35 fleet and balanced for their impact on overall system change capacity of the F-35 enterprise. Ultimately, the effect of any software change will be evaluated in terms

of net worth provided to the warfighter.

F-35 block updates will *bundle* the delivery of new functionality and PBO maintenance changes. Block updates will include a mix of all types of software changes, and may encompass hardware/subsystem changes. As IPTs develop and produce the changes to support F-35 block plans, capacity planning must allow for *both* PBO changes and for new functionality. Accordingly, the processes, definitions, requirements and practices for software *maintenance* planning were merged with the F-35 template for software development plans.

On the plus side, technical solutions are not constrained. System changes, hardware changes and software changes, corrections, and new functionality are all assessed with respect to the end effect on air system performance and affordability.

On the downside, cost accounting within IPTs responsible for producing corrective changes and new functionality requires extreme fidelity to ensure PBO effort is distinguished from follow-on development.

5. Develop Highly Maintainable Systems and Software

Maintainability of F-35 software is based upon an AV with an open and scalable architecture. The *open architecture* allows for expansion with minimal impact to unchanged elements, through use of well-defined, non-proprietary interfaces and protocols. Hardware and software elements are partitioned using loosely coupled, non-time critical interfaces. A *data collection domain*, added to the AV architecture, supports general instrumentation and fault isolation requirements. An *isolation layer* protects the software investment from hardware obsolescence and facilitates multi-use across air system domains.

DOORS (Dynamic Object Oriented Requirements System) databases are populated with requirements and the rationale for their selection, including linkages to the architectural models. Modeling and simulation tools based on architectural constructs are employed to develop and validate the requirements and verify the air system.

Object-oriented design results in smaller configuration items with clearly defined functionalities. *Hardware independence* supports problem accountability and localizes change. This approach results in minimal changes to the overall configuration when adding or deleting functional capability. It also supports change development at the lowest level, minimizes the impact of changes, and enables *focused testing. Focused, model-based component testing* provides opportunities for efficiency in an area that typically entails high cost.

Architectural and design integrity is maintained through use of structured, *common systems engineering processes and tools*, which have reduced initial development costs and will support the efficient long-term maintenance of designs. The JSF Systems/Software Engineering Environment (S/SEE) resides on networks of computing equipment which connect F-35 customers, contractors and subcontractors. This shared environment is used across the entire team to foster a unified understanding of the open architecture and its maintenance. The S/SEE includes commercial off-the-shelf (COTS) Unified Modeling Language tools, such as Rhapsody, for enforcing the object-oriented software design and the Signal Interface Management System[2] (SIMS) tool, which forces full interface definition.

Autocoding with JSF S/SEE tools allows software code to be developed to Open System Architecture (OSA) standards. In specific domains, design tools are used to capture requirements, model a functional algorithm and provide source code as an output to implement the modeled design.

Emphasis is placed on leveraging COTS, and *supportability* (aggressive preparation for tool obsolescence). Configurations of S/SEE tools are managed in consonance with air system software releases to ensure build repeatability and maintainability.

Maintainability of F-35 software is also supplemented by software *reuse and multi-use.* (Reuse is the use of pre-existing code; multi-use is the reuse of code in multiple areas.) Software with proven maturity and reliability, from various sources including legacy aircraft, government-furnished equipment, software and databases, different F-35 variants, off-the-shelf software, subsystem software, and third party suppliers, is reused to the extent possible. Reuse objects are also used as starting points for development of other, closely related objects that may be required for different F-35 domains. Availability and quality of documentation and source files are considered as part of reuse analysis and determination. *Preconditioning* of software targeted for reuse may be performed as necessary in order to ensure maintainability. If candidate code does not meet OSA standards or object orientation, but provides needed functionality with an effective design, the design and algorithms (only) are reused and recoding is performed using architectural patterns to develop software which provides better long-term maintainability and lower total ownership cost than pre-existing software that is *wrapped* to be OSA compliant.

6. Manage Off-the-Shelf Software

The up-front savings realized through use of off-the-shelf software is frequently offset by risk and expense incurred later in the product life cycle. (Off-the-shelf software includes COTS, modified off-the-shelf [MOTS], government off-the-shelf [GOTS], freeware, public software, and related categories of non-developed software.) Accordingly, special emphasis was placed on managing off-the-shelf software. Through a collaborative effort with the JPO, a process document entitled, "Off-The-Shelf Software in the JSF Software Lifecycle" [3] was produced and deployed. The process document describes actions, over and above standard software process requirements applicable to software developed specifically for the JSF Program, which must be taken to ensure off-the-shelf software components are configuration managed throughout their life cycle. It also identifies the functions/roles responsible for those actions. Instructions in the process are partitioned according to their applicability to a 4-phase life cycle, along with generally applicable rules, guidelines, and warnings.

Based on results of a 2005 collaborative evaluation of process implementation, "Off-The-Shelf Software in the JSF Software Lifecycle" was revised and updated to include a system for classification of off-the-shelf software projects (small, medium, or large projects) based on specified criteria. Suggested tailoring of process requirements based on project category is included along with examples. The revised process also incorporates a requirement for a generation of a compliance matrix by off-the-shelf software projects, to ensure that all applicable requirements, including license and distribution controls, are adequately addressed.

7. Plan for the Unexpected

Warfighters are keenly interested in how the F-35 AL global sustainment solution will respond to urgent operational requests, or to emergencies. To answer these concerns, software sustainment scenarios have been developed by the F-35 SMS WG. The scenarios contain sufficient detail to describe the activities required for non-routine situations. The scenarios are used to exercise ARIS process models and make an up-front determination of the cost and time required to perform all needed activities.

8. Analyze and Refine the Software Sustainment Business Case

The eight steps featured in this article have covered a lot of territory. But the steps

begin and end with a focus on money. Step 1 focused on achieving affordability through commonality. This final, 8th step addresses the business case analysis of F-35 sustainment, annual global sustainment total ownership cost estimates, and the software cost estimation practices that support these analyses. Each of these interconnected activities uses a spiral development approach with each spiral providing increased fidelity of data, inclusion of decisions from across the program, updates on configurations and reliability projections, and comprehensive detailing of the business offering.

Business Case Analyses (BCAs) define F-35 global sustainment policies and processes. These analyses answer three basic questions: 1) What individual tasks must be performed during sustainment?, (2) who (government or contractor) should perform those tasks, based on *best value?*, and (3) does the task allocation support established performance standards and provide sufficient savings? BCAs address, for example, a pricing architecture for PBO sustainment, international taxes and tariffs, industrial base capacity and responsiveness, and, of course, software sustainment.

Global sustainment cost estimates are formulated annually. These estimates integrate estimates from all IPTs, functions, team member companies, and subcontractors involved in the F-35 global sustainment solution and the pilot program for performance-based sustainment. Annual cost estimates are consistently produced, fact-based, and supportable. They are reconciled with the JPO affordability cost analysts and are finalized and formally approved at JSF cost summit events. The integrity of these annual global sustainment cost estimates is critical to the success of affordable, PBO sustainment of the F-35 fleet.

Parametric software sustainment cost estimates are developed for inclusion in the annual global sustainment estimates using output from the System Evaluation and Estimation of Resources – Software Estimating Model (SEER-SEM) tool. Software sustainment cost estimates align with Cost Analysis Improvement Group Element 6.5, "Software Maintenance Support." They are not, however, fully representative of all costs associated with performance-based software sustainment and are subject to ongoing refinement and updates. Updates provide greater detail and direct estimates for software integration and test activities, software lab *keep warm* costs, and greater fidelity with respect to license costs for off-the-shelf software included in deliverable F-35 software products. In the absence of *actual* data, ground rules and assumptions are documented and version-controlled to describe cost areas which are included in, or excluded from, the F-35 software sustainment business case.

Finally, long-term software sustainment cost estimates entail software maintenance *and* software growth estimates. Once a software release (which, as we have seen, will contain fixes and new functionality) is distributed to the field, it becomes the new *maintenance baseline* and PBO contracting for when the new release takes effect.

Conclusion

The decision to apply a performance-based sustainment approach to the F-35 has caused fundamental changes in the approach to Air System sustainment. Traditional roles and responsibilities are shifting. An increased risk is transferred to contractors who are now responsible for system availability and mission success. This has precipitated a new approach to software sustainment. While results are years away, the F-35 software community has put a foundation in place for PBO software sustainment. Construction on that foundation continues, day by day.◆

References

1. Christie, Rebecca. "Lockheed Martin Hopes F-35 Leads To Maintenance Revolution." DOW Jones Newswires 12 June 2007.
2. IEEE. "IEEE Standard for a High Performance Serial Bus." IEEE 1394, 1995. "Amendment One." IEEE 1394a, 2000. "High-Speed Supplement."IEEE 1394b, 2002.

Notes

1. Fifth Generation Fighter features these attributes: advanced stealth, information fusion, high agility, enhanced situational awareness, new levels of reliability and maintainability, and network-enabled operations.
2. SIMS is a Lockheed Martin Aero internally developed interface management tool, based on a commercial relational database and used on multiple aircraft platforms.

About the Authors

Lloyd Huff is a Lockheed Martin senior fellow for software and avionics where he is currently engaged in the development of the F-35 software sustainment solution. Recently, Huff served as director of JSF Software Management, JSF Software Proposal Lead, and X-35 STOVL First Flight Deputy Lead. He has 28 years of aerospace experience including F-16 multiplex buss design, LM Site Lead at Hill AFB, 50% Software Cost Reduction Lead, CMM Level 4 Tool Development Lead, and Principle Investigator for Multiplex and Fiber Optic Research. Huff holds bachelor's and master's degrees in computer science from the University of Kansas.

Lockheed Martin Aeronautics
P.O. Box 748, MZ 1523
Fort Worth, TX 76101
Phone: (817) 821-9196
Fax: (817) 763-1737
E-mail: lloyd.a.huff@lmco.com

George Novak is a Lockheed Martin software senior staff member where he is responsible for planning the long-term global sustainment of software under a performance-based logistics contracting model. Novak is a member of the software management team responsible for air system software builds and software life-cycle planning for the F-35 air system, and is co-chairman of the F-35 software maintenance and sustainment working group. He has held a variety of positions on military aircraft and commercial telecommunications programs and has served as a lecturer and independent consultant in the area of software process improvement. Novak has a master's of business administration in industrial management from the University of Dallas.

Lockheed Martin Aeronautics
P.O. Box 748, MZ 2306
Fort Worth, TX 76101
Phone: (817) 763-3863
Fax: (817) 763-1737
E-mail: george.j.novak@lmco.com

Reference Metrics for Service-Oriented Architectures

Dr. Yun-Tung Lau
Science Applications International Corporation

This article presents a set of reference metrics for measuring the quality of services in Service-Oriented Architectures (SOAs). It introduces the metrics cube and scalability curve and applies them to the development of service-level agreements (SLAs) and capacity planning. This article also discusses the challenges and approaches for defining and allocating end-to-end metrics in a net-centric environment such as the Global Information Grid (GIG).

In an SOA, a set of loosely coupled services work together over a network to provide functionalities to end-users [1]. The service provider registers information about a service at a service registry. Service consumers can find the service from the registry and then invoke the service through the service interface.

For the Department of Defense (DoD), a set of GIG Enterprise Services will provide warfighting, business, and intelligence capabilities to support operational missions conducted by various communities of interest [2]. Examples include Net-Centric Enterprise Services (NCES) [3] and Net-Enabled Command Capability (NECC) [4].

Services in an SOA have well-defined service interfaces. They also have SLAs which are parts of the service contracts that specify the levels of service expected after deployment. A key aspect of an SLA is the set of metrics for measuring performance and quality of service. This article develops an overarching model of reference metrics relevant to end-user experience. It introduces the concept of a metrics cube that captures the relationship between the metrics which are then applied to the development of SLAs and capacity planning.

The reference metrics are important to the successful implementation, deployment and sustainment of SOA in the GIG because of the following:

- They form the basis for combining metrics across network and computing infrastructures for services in the GIG net-centric environment. Since those infrastructures typically fall under various responsible entities, having a basic reference set is critical to the development of end-to-end metrics relevant to an end-user's experience.
- They relate directly to consumer's (end-user) experience using a service. This includes timeliness, scalability, availability, and reliability, which are specified in SLAs.
- They are used throughout a system engineering life cycle, including requirement definition, SLA development, service design, performance testing, and SOA sustainment.

Reference Metrics

The reference metrics are collectively referred to by their symbols as the TSAR (service Time, Scalability, Availability, Reliability) metrics. They are defined in more detail in Table 1.

For synchronous services, such as a request/response Web service, T is simply the response time. It is measured from the time a consumer sends a request to when the consumer receives a response. Typically, T will have an average and a standard deviation. It is the sum of network latency (including transmission time, propagation time, Internet protocol delay, and congestion) and time spent at the service provider (including local processing time and back-end processing time).

"The TSAR metrics relate directly to consumer (end-user) experience with a service by answering the following questions: how fast (T), how much/many (S), how durable (A), and how reliable (R)."

For asynchronous services, such as a messaging service, T is the delivery time. It is measured from when a publisher sends a message to when subscribers receive it. T is typically a distribution with T_min, T_average, and T_max. A service-level agreement may guarantee delivery within a certain T_max.

Scalability (S) measures a service's ability to handle growing amounts of work within the desired time and reliability ranges. Examples are user load (number of users within a certain time span), number of requests per unit time, and size of requests or messages over a certain time.

Availability (A) is defined as one minus the percentage of planned and unplanned service down time. In other words, it is the combined probability that a service is up and running. It is often expressed as a number of nines, such as 99.9 percent (8.8 hours down/year). Contribution for A comes from planned hardware/software maintenance, hardware failure of networks and processors, and software failure due to fatal defects (e.g. memory leaks).

Finally, Reliability (R) is the percentage of service completion with anticipated results when the service is *available*. Hence if R = 95 percent, the error rate is 5 percent. Errors generally come from non-fatal software defects, requests rejected by load control mechanism (when availability is within the required range), or message loss during delivery (e.g., due to congestion or faulty network hardware). Unexpected results caused by problems in back-end processing (e.g., time out or failure of dependent services) are also considered errors. Note that reliability is defined at the application level. It measures how reliable a service performs its function when it is up and running.

The TSAR metrics relate directly to consumer (end-user) experience with a service by answering the following questions: how fast (T), how much/many (S), how durable (A), and how reliable (R).

Metrics Cube

The TSAR metrics are not all independent. In general, as *S* increases, *T* goes up, and R goes down. The plot of T or R versus S is called a *scalability curve*.

Metrics	Symbols	Notes
Service time	T	Response time for synchronous services. Delivery time for asynchronous services.
Scalability	S	Examples are user load and number of requests per second.
Availability	A	It includes planned maintenance and unplanned down time.
Reliability	R	Due to defects, rejected requests, message loss, etc.

Table 1: *Reference Metrics for SOAs*

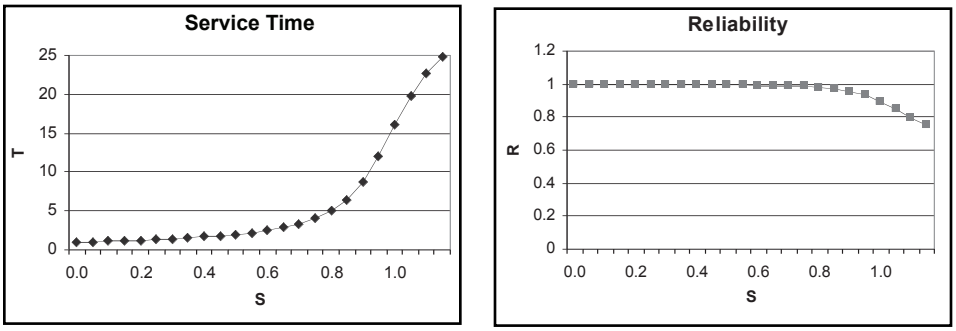


Figure 1: *Scalability Curves*

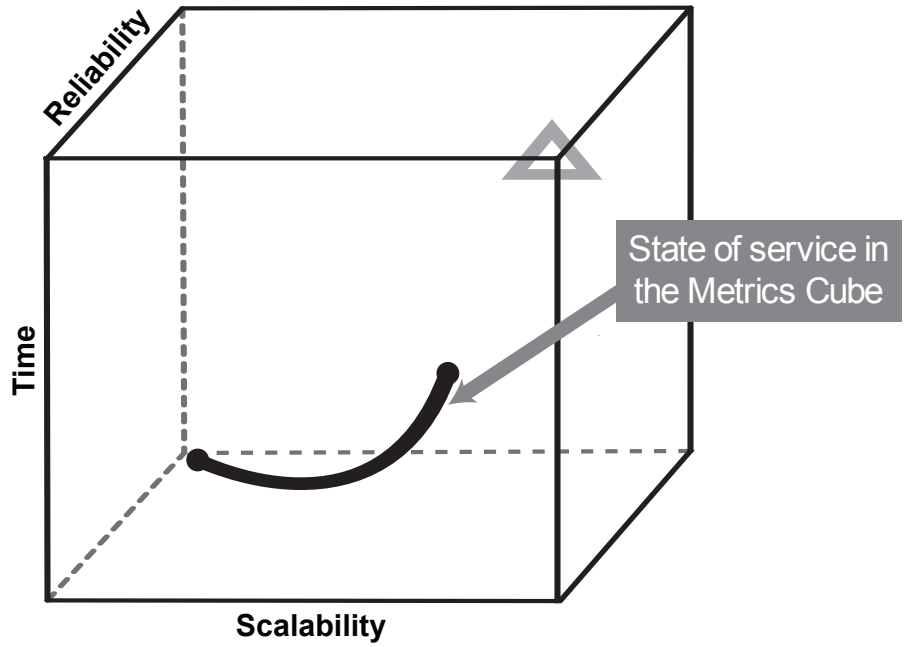
Figure 1 shows an example based on a model with a finite service queue [5]. As S increases, incoming requests/messages spend more time waiting in the queue, causing T to increase. Network latency is not included in this example. Also, in Figure 1, R includes the effect of both software defects and rejected requests (the latter happens when the number of requests in the queue reaches an upper limit). The exact values of the curves in Figure 1 are not important for the discussion here. An appendix in the online version of this article provides further details about the model.

In Figure 1, S is the rate of service requests relative to the maximal throughput μ. Here throughput is defined as the number of completed requests per unit time. As S increases from zero, the throughput increases linearly with S. However, as S approaches one (the rate of service requests approaching the maximal throughput), the throughput plateaus at μ and an increasing fraction of incoming requests are dropped. This is due to the limited computing or network infrastructure supporting the service. The finite queue model simulates this effect by limiting the number of requests/messages in the queue. Below the maximal throughput, one may use throughput as an approximate measure for S. This is convenient since commercial tools typically provide throughput as one of the measures.

The contributing factors to availability do not change as S increases. Thus, availability can generally be considered a constant. However, at very high levels of S, the extremely high demand exhausts the underlying computing and network infrastructure for the service, making it unable to perform any work (similar to a denial-of-service attack). This leads to an abrupt drop in availability. For all practical purposes, it should be determined experimentally that this situation does not occur within the expected range of S. Once this is done, availability can be considered independent of S. The following discussion assumes that this is true.

To help visualize the scalability behavior of a service, one may define a *Metrics Cube* using the minimum and maximum values of S, T, and R. The boundary S_min represents the threshold value and corresponds to the lower bound of normal operation. S_max, on the other hand, is the objective or peak operation value. As S increases, the state of a service can be traced along a scalability curve in the cube, as shown in Figure 2. The metrics cube is useful for specifying SLAs. This is discussed in the next section.

Figure 2: *Metrics Cube*



SLAs

The essence of an SLA is to specify a metrics cube and a required availability (A) range, as well as the statistical calculation of the metrics (e.g. average over an hour, one day, etc.). A nominal process of developing an SLA follows:

1. Service provider measures the scalability curve for a service.
2. Service consumers submit service-level requirements, which can be expressed as a metrics cube and availability range.
3. Service provider compares scalability curves with the requirements.
4. Service provider adds or subtracts computing resources to do the following:
   a. Optimize the scalability curve within the metrics cube.
   b. Meet the desired availability range.

5. Service provider refines and negotiates the SLA with the consumers (based on cost, schedule, and other factors).

In step 4, the scalability curve is shifted within the metrics cube when computing resources are changed. Figure 3 shows a cross-section of the T and S plane in a metrics cube. If the upper end of the scalability curve touches the T_max boundary, the SLA may potentially be violated because the service time will exceed the allowed maximum before S_max is reached. By adding computing resources (e.g. more servers), the curve is shifted downward.

However, if overdone, the curve comes well below T_max at the S_max boundary, indicating over-engineering and wasted computing resources; thus, the optimal configuration is to have the curve touch the upper corner (or somewhat below it as a reserve). Similarly, the optimal curve for reliability should touch the corner at (S_max, R_min). In terms of the metrics cube in Figure 2, the optimized scalability curve would touch the corner labeled with a triangle.

To be compliant with an SLA, the service provider needs to ensure that availability is within the required range. When the service is up and running, the service provider monitors the metrics and ensures that they stay within the required metrics cube.

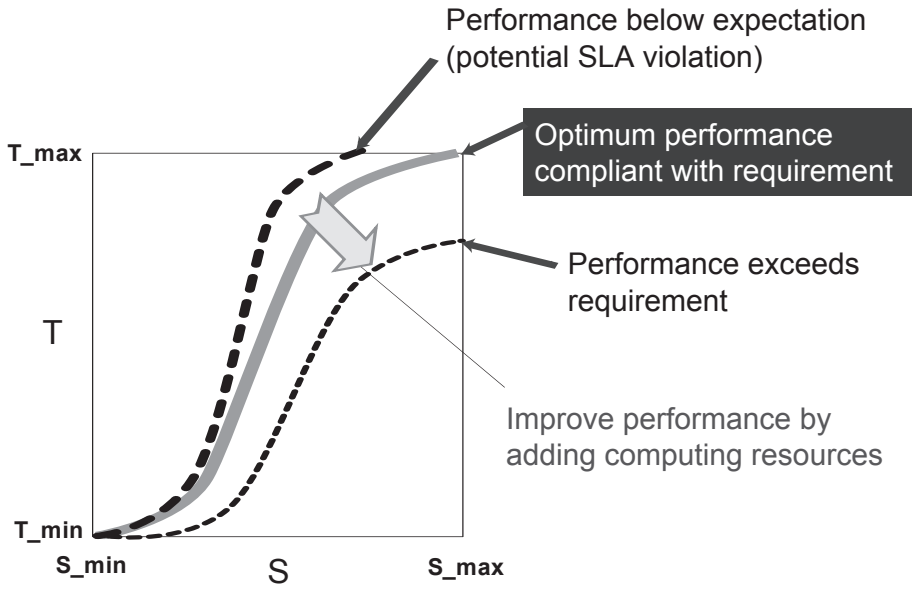


Figure 3: *Optimization of Scalability Curve*

Closing Remarks

This article defines a set of four reference metrics (collectively called TSAR metrics) for measuring the quality of sustained services in SOAs. It introduces the concept of a metrics cube, which is applied to the development of SLAs and capacity planning of computing resources.

For example, for NCES, a set of threshold and objective metrics have been defined. They are the equivalent of the minimal and maximal boundaries of the metrics cube. For NECC, the TSAR metrics are included in the developer's guide [6] and used in the system engineering process, most notably for SLA development.

An inherent challenge for defining end-to-end metrics (such as the TSAR metrics in Table 1) is that they typically have distributed contributions across network and computing infrastructures. Hence the responsibility for ensuring SLA compliance is shared by multiple entities in a net-centric environment such as the GIG. Nevertheless, the service provider is normally the primary sustainment interface to its service consumers. The provider allocates metrics to its dependent network and computing service providers, either through subordinate SLAs or by explicitly allocating portions of a metric to their responsible entities. The bases of such allocation are the formulas for combining metrics from multiple contributors (e.g. service providers, infrastructures). An appendix in the online version of this article provides such formulas.◆

Acknowledgment

The comments of John Schumacher (Principal Systems Engineer at SAIC) on an early draft of this article are greatly appreciated.

References[1]

1. Erl, Thomas. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005.
2. United States. Joint Forces Command (JFCOM). "Global Information Grid Capstone Requirements Document." JFCOM 134-01. JFCOM, 2001.
3. United States. Defense Information Systems Agency (DISA). "Initial Capabilities Document for Global Information Grid Enterprise Services." DISA, 2004.
4. Assistant Secretary of Defense, Networks and Information Integration. "Net-Enabled Command Capability Milestone A Acquisition Decision Memorandum." Washington: DoD, 2006.
5. Menascé, Daniel A., and A.F. Virgilio. "Capacity Planning for Web Services: Metrics, Models, and Methods." Prentice Hall, 2001.
6. "Net-Enabled Command Capability Developer's Handbook." DISA, 2007.

Note

1. Some Web sites quoted here require a user account for access. Online application forms can be found on the sites. Some require government sponsorship.

About the Author

Yun-Tung Lau, Ph.D. is Vice President of Technology at SAIC. He has been involved in large-scale software architecture, design, and development for 18 years. Lau has served as chief architect for many software and enterprise architecture projects, from scientific computing and electronic commerce to command and control systems. He also holds a master's degree in technology management. Lau has published many articles in professional journals, and authored the book "The Art of Objects: Object-Oriented Design and Architecture."

SAIC
5113 Leesburg Pike
STE 200
McLean, VA 22041
Phone: (703) 824-5817
Fax: (703) 824-5836
E-mail: yun-tung.lau@saic.com

Coming Events

January 7-10
Hawaii International Conference on System Sciences
Waikoloa, HI
www.cert.org/secure-coding/HICSS-41-CSSIS_CFP.html

January 10-12
The 35th Annual ACM SIGPLAN – SIGACT Symposium
San Francisco, CA
www.cs.ucsd.edu/popl/08/

January 10-12
The 5th IEEE Consumer Communications and Networking Conference
Las Vegas, NV
www.ieee-ccnc.org/2008/

January 14-16
Soldier Technology US
Arlington, VA
www.soldiertechnologyus.com

January 22-25
Network Centric Warfare 2008
Washington D.C.
www.ncwevent.com

January 30-31
15th Annual Multimedia Computing and Networking
San Jose, CA
http://mirage.cs.uoregon.edu/mmcn2008/

April 29-May 2
Systems and Software Technology Conference
Las Vegas, NV
www.sstc-online.org

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

Web Sites

Scientific and Technical Information Network
http://stinet.dtic.mil
The Public Scientific and Technical Information Network (STINET) is available to the general public, free of charge. It provides access to citations of unclassified unlimited documents that have been entered into the Defense Technical Information Center's Technical Reports Collection, as well as the electronic full-text of many of these documents. Public STINET also provides access to the Air University Library Index to military periodicals, staff College Automated Military Periodical Index, Department of Defense (DoD) Index to Specifications and Standards, and Research and Development Descriptive summaries.

The Data and Analysis Center for Software
www.thedacs.com
The Data and Analysis Center for Software (DACS) is a DoD Information Analysis Center. The DACS has been designated as the DoD Software Information clearinghouse that serves as an authoritative source for state-of-the-art software information providing technical support for the software community. The DACS technical area of focus is software technology and software engineering, in its broadest sense. The DACS is a central distribution hub for software technology information sources. The DACS offers a wide variety of technical services designed to support the development, testing, validation, and transitioning of software engineering technology.

Java.net
www.java.net
Java.net is the realization of a vision of a diverse group of engineers, researchers, technologists, and evangelists at Sun Microsystems, Inc. to provide a common area for interesting conversations and innovative development projects related to Java technology. The community continues to grow with industry associations, software vendors, universities, and individual developers and hobbyists joining every day. As they meet, share ideas, and use the site's collaboration tools, the communities they form will uncover synergies and create new solutions that render Java technology even more valuable.

A Primer on Java Obfuscation

Stephen Torri, Derek Sanders, and Dr. Drew Hamilton
Auburn University

Gordon Evans
Missile Defense Agency

Java is not a secure language and its increasing use puts sensitive information at risk. While the authors do not recommend Java software that involves sensitive information, the current reality is that Java is used in these applications. To address this reality, this article discusses Java obfuscation techniques.

In today's software-oriented world, software ownership frequently changes. It is difficult, if not sometimes impossible, to keep track of who has a certain piece of software at any given time. This presents a problem for those who wish to keep the software internal operations a secret. Languages such as Java, which preserves a lot of high-level information in its byte code, present a problem from the standpoint of source ownership and securing it from program de-compilation. Releasing Java classes can compromise sensitive information embedded in the software, such as a missile intercept computation. Java class files contain the byte code instructions interpreted by the Java Virtual Machine (JVM). These files are easily read by programs that can recreate a source file from the class file. Java obfuscation techniques were developed to make reverse engineering harder, but many of these techniques can be defeated.

While Java was developed to be used on embedded systems, its popularity has pushed it into the public as a mainstream language. It is the view of the authors that if the developers of a program (e.g. defense industries) do not want its source code reverse engineered, then Java should not be used. Java programs cannot be protected in any manner from reverse engineering. All protection will do is slow down a determined attacker. In this article, we describe the three major techniques of Java obfuscation used in present state-of-the-art tools.

Commercial obfuscation applications generally perform three functions to secure the Java source code. First, extra loops, jumps, or even additional classes are added to change the control flow of the program so that an attacker has extra difficulty in understanding the program. Second, Package/Class/Method/Field names are renamed so they no longer state what they are for (e.g., field named 'account_balance' is now 'b'). Finally, any text strings contained in the program are encrypted.

Obfuscation Techniques

The following sections describe the three major techniques of Java obfuscation used in present state-of-the-art tools.

Control Flow Obfuscation

Control flow obfuscation is a technique that makes use of additional code and looping it to make it difficult to understand what is going on in such a way that causes an attacker to give up or confuses a tool into producing undesired results. While this strength is a good attempt at protection, there are semi-automated tools, such as LOCO [1], that allow a human user to interpret the code to distinguish between useless code and real code. While control flow obfuscation is not foolproof, it increases the difficulty an attacker has reverse engineering a program.

Control flow obfuscation is a technique that makes use of additional code and looping it to make it difficult to understand what is going on in such a way that causes an attacker to give up or confuses a tool into producing undesired results.

Name Obfuscation

Name obfuscation is used to effectively remove any information an attacker would gain by merely reading the name of fields. For example, if the original developer used meaningful names to aid development, this would also help the attacker. By changing the names, the meaning of the code is harder to understand. This is quite similar to the problem of decompiling x86 binaries. When decompiling x86 binaries into an intermediate language, e.g. Register Transfer Language, an attacker has to figure out the contents of the accumulator register and how it is used. This can be extremely tedious but not impossible. Similarly, an attacker with a Java class file, where the names are changed to simple letters (e.g. 'b' or 'c1'), is faced with a similar challenge. The strength of this method is that it removes a very useful method of program comprehension from the hands of an attacker. However, its weakness is that a human using an interactive deobfuscation environment, possibly a modified LOCO equivalent program, can discern what the variable 'b' means, and the program they use could propagate this new name 'account_balance' throughout the control flow graph where 'b' is used. Name obfuscation raises the level of difficulty in reverse-engineering, but does not make it impossible.

String Encryption

String encryption is utilized as an attempt to secure the code for a limited period of time. The more sensitive the information being protected, the stronger the encryption should be. By eliminating another source of information, obfuscation programs use this technique to increase the level of difficulty in an attempt to prevent deobfuscation. However, string encryption is almost useless since the key for decryption is contained inside the program file unless using an external key. It has been shown that attackers have already discovered how to decrypt these strings [2], rendering this obfuscation technique almost useless. Encryption is useful only if an external key is used. This, however, presents the classic key distribution and management issue. Using an external key requires securely sharing it via some mechanism, which is outside the scope of this article.

JAVA Byte Code

Java is compiled from source files into class files containing byte codes that are later interpreted or compiled into machine code at runtime in a JVM. The Java class files present a potential security problem since simply compiling the Java source code does not do enough to secure it from being recovered. Disassembly of the Java byte code is easy to do with the tools provided by Sun Microsystems as a part of its software development kit (SDK). For example, the following is the classic *Hello World* written in Java:

```
public class Hello {
public static void main ( String[]
   args )
  {
   System.out.println("Hello
   World");
  }
}
```

This example simply prints out the string saying *Hello World* to the standard console window. Compiling this program with the javac compiler produces a Java class file called Hello.class. This file is used with the Java program to produce the desired results. The Java class file can be easily disassembled into a human readable form using the javap [3] disassembler program included in the Sun Microsystems Java Development Kit (JDK).

For example, the following is the output of Hello.class after running javap:

```
Compiled from "Hello.java"

class Hello extends
   java.lang.Object {

Hello();
Code:
0: aload_0
1: invokespecial #1; //Method
   java/lang/Object."<init>":()V
4: return

public static void
   main(java.lang.String[]);
Code:
0: getstatic #2; //Field java/
   lang/System.out:Ljava/io/
   PrintStream;
3: ldc  #3; //String Hello World
5: invokevirtual #4; //Method java/
   io/PrintStream.println:
   (Ljava/lang/String;)V
8: return
}
```

We have recovered enough information that a developer with a tool such as the Dava decompiler, (McGill University's Java decompiler) included in the Java optimization framework called Soot [4], can quickly obtain the original source code seen in the following:

```
import java.io.*;

class Hello {

  Hello() {
     super();
  }

  public static void
   main(java.lang.String[] r0) {
       System.out.println
          ("Hello World");
  }
}
```

This example is a simple one but it illustrates the point. It can be seen that by merely compiling a Java application with the Sun JDK will not offer any protection against decompiling the program. This is why developers use obfuscation in order to get some level of protection against reverse engineering. Obfuscation makes it harder, but not impossible, to reverse engineer the code.

JAVA Obfuscation

Obfuscation works by confusing the flow of the source code so it is difficult to recover the intent of it. However, in order to effectively show how obfuscation works, a complex example is needed. The following code is a function that takes an integer value from the command line as an argument and reports back the list of Fibonacci[1] numbers. For example, running the command *java Fibonacci 5* will give back the calculated Fibonacci number for 5, 4, 3, and so on.

```
public class Fibonacci {

public int calculate (int n)
  {
     int output = 0;

     if (n > 1)
     {
       output = calculate (n - 1)
          + calculate (n - 2);
     }
     else
     {
        output = n;
     }
     return output;
  }
public static void main ( String[]
   inc )
   {
     if (inc.length > 0)
    {
       Fibonacci f_ref = new
          Fibonacci();
       int n =
          Integer.parseInt(inc[0]);
       while ( n != -1 )
   {
     System.out.println
       ("Calculated fibonacci
        number: " +
        f_ref.calculate ( n ) );
     n-;
   }
   }
   return;
   }
}
```

After using javap, the byte code prior to obfuscation is shown in the following:

```
public int calculate(int);
Code:
0: iconst_0
1: istore_2
2: iload_1
3: iconst_1
4: if_icmple 26
7: aload_0
8: iload_1
9: iconst_1
10: isub
11: invokevirtual #2; //Method
    calculate:(I)I
14: aload_0
15: iload_1
16: iconst_2
17: isub
18: invokevirtual #2; //Method
    calculate:(I)I
21: iadd
22: istore_2
23: goto 28
26: iload_1
27: istore_2
28: iload_2
29: ireturn
```

The commercially available obfuscation program called Zelix Klassmaster is used to obscure the names of classes, methods, and variables; encrypt any strings; and complicate the control flow. Though the nature of the program is hidden and obscured, the byte code is still easy to read. The important blocks of obfuscated byte code are explained in the following:

```
public int a(int);
```

```
Code:
0: getstatic #56; //Field A:Z
3: istore_3
```

The previous lines are loading the value of a static variable from the class A, called Z, onto the stack. The value is stored into the third local variable (var_3 = A:Z).

```
4: iconst_0
5: istore_2
```

These instructions set the second local variable to zero (var_2 = 0).

```
6: iload_1
```

This instruction loads the value of function parameter 'n' onto the stack.

```
7: iload_3
8: ifne 50
```

These instructions are checking to see if var_3 (the third local variable) is not equal to zero. If the statement returns true then it will jump to label #50, otherwise it continues to label #11. Though not seen here, at label #50 the variable 'output' is set to the value of variable 'n' and returned.

```
11:     iconst_1
12:     if_icmple 49
```

At this point, the constant integer value of '1' is loaded onto the stack, which is used to compare the value of the previous stack entry 'n' to 1. If 'n' is less than 1 then it will jump to label #49, otherwise it continues to label #15. Label #49 is not shown, but its instruction sets the variable 'output' equal to the value of variable 'n' and is returned. The two checks at lines 7-8 and 11-12 that were performed are different from the original check in the unobfuscated code to see if variable 'n' was greater than 1. The obfuscation program has altered the control flow in an attempt to obscure the nature of the function.

```
15: aload_0
16: iload_1
17: iconst_1
18: isub
19: invokevirtual #2; //Method
    a:(I)I
```

The function a:(I)I is the original function called calculate (int n) that returns an integer result. This byte code loads an object reference to the variable n, the value of variable n and a constant integer value of '1' onto the stack. It then calculates n-1 and places the result on the stack. The call to the function a:(I) with the results is the last step. This is equivalent to the function call of 'calculate (|n – 1|).'

```
22: aload_0
23: iload_1
24: iconst_2
25: isub
26: invokevirtual #2; //Method
    a:(I)I
```

These instructions are similar to the description above, except the function call is equivalent to 'calculate (n – 2).'

```
29: iadd
30: istore_2
```

At this point the results of 'calculate (n – 1)' and 'calculate (n – 2)' are taken from the stack, added together and the result is placed back on the stack. This is similar to 'calculate (n – 1) + calculate (n – 2)'. The results are stored in var_2.

```
31: iload_3
32: ifeq 51
35: getstatic #58; //Field z:Z
38: ifeq 45
41: iconst_0
42: goto 46
45: iconst_1
46: putstatic #58; //Field z:Z
```

Shown here is a reference to the variable Z from class z. However, notice that the original program did not contain a second class, but the obfuscator has added it to obscure the meaning. Labels #31-32 compare the value of var_3 to zero. If var_3 is equal to zero then the value of var_2 (original variable called 'output') is returned, otherwise, the comparison of the variable 'Z' from the class 'z' is compared to zero. If 'Z' is equal to zero then the value of 'Z' is set to 1, otherwise zero. These instructions are inserted by the obfuscator as *do nothing* statements to enhance the security and complicate deobfuscation forcing additional work to obtain the original code.

```
49: iload_1
50: istore_2
51: iload_2
52: ireturn
```

Finally, the results of the function call are returned to the original caller.

Even with obfuscation, anyone with access to the Java class files has access to the byte code and hence is capable of reversing the obfuscation process. The LOCO project, which is designed to aide a security analyst in understanding obfuscated code, could be used for this purpose. While the project is designed to look at instructions on an x86 architecture, a similar project designed for Java byte code would be much simpler to implement. This is due to the fact that the number of instructions that are represented by Java byte code is considerably less than the number of instructions for the x86 architecture. The weaknesses of obfuscation as shown with these simple examples illustrate the need for better protection against reverse engineering. In addition, the impact of obfuscation has on the performance of the software must also be analyzed and evaluated for acceptability. While many, if not most, Java developers do not read Java byte code, a determined adversary can and will.

Cost of Obfuscation

In order to effectively discuss obfuscation, the impact of obfuscation on performance with normal operations can not be ignored. Low [5] states that obfuscation should not alter the behavior of the program, which is shown next:

Obfuscating Transformation
Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{T} P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $P \xrightarrow{T} P'$ to be a legal obfuscating transformation the following condition must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

The authors believe that changes to the program's control flow and the use of string encryption will inadvertently affect software performance. The degree of the impact depends on the control flow obfuscation method and encryption algorithm used. The effect of name obfuscation does not impact the run-time performance of the system. To better understand the impact of obfuscation, it must be shown in terms of runtime in a formal manner.

Control Flow Obfuscation

Intuitively, obfuscating the control flow of a Java program should incur some performance cost as it is interpreted. Definition

1 defines a performance measure for control flow obfuscation delay.

Definition 1: Control Flow Obfuscation Delay

Let $T_{ocf} = T_{cf} + \alpha$ be an equation showing the effects of obfuscation T_{ocf} on original system performance T_{cf} by time delay of control flow obfuscation α. If $T_{ocf} \leq T_{cf}$, then the obfuscation has either improved the original performance of the program or, at a minimum, met the original performance. More accurately, the equation is $T_{ocf} = T_{cf} + \alpha$, where $\alpha \leq 0$. Alternatively, if α is greater than zero, then the obfuscation has had a negative effect on system performance.

An embedded system may have hard real-time constraints which restrict how much additional delay is allowed. By real-time we refer to systems which will fail if the executing software should miss a deadline. The impact of obfuscation on the execution of the program would need to be measured – α in the equation above – to determine if it is at an acceptable level that does not degrade the system performance or user experience. That is, if $T_{ocf} > T_L$, where T_L is a limit of a real-time deadline or acceptable delay, then control-flow obfuscation may produce more harm than good.

String Encryption

String encryption on the other hand will definitely not have an obfuscation effect of zero. In the programs evaluated in [2], three of them utilized string encryption. The key used was stored in the program file along with the decryption code. The encrypted strings were either kept in the program's class files or had extra files included in the Java jar file.

The time delay caused by decryption depends on the encryption algorithm, key length, and the plain text. The original program had to access the location in memory, where the original string was, and return it to the place in the program it was used ($\delta = T_r$, where T_r is the retrieval time). Compare this to the time it takes to retrieve the encrypted string, perform the decryption algorithm, and return the plain text string ($\delta = T_{er} + T_d + T_{pr}$) where T_{er} is the time to retrieve the encrypted string, T_d is the decryption time, and T_{pr} is the time to return the string to the requester. Therefore, the time required to process and return the encrypted string should be greater than that of a non-encrypted string.

Definition 2: Encrypted String Obfuscation Delay

Let $T_{es'} = T_{ps} + \delta$ show the effect of using encrypted strings, T_{es}, on system performance using plain strings, T_{ps}, by the time delay for encrypted string decryption, δ. Then the time delay of decryption should never be zero ($\delta > 0$), therefore, $T_{es} \neq T_{ps}$, since the act of decryption is not an act that cannot be simply dismissed as some that can be ignored. Some amount of time would be required so it is more accurate to say $T_{es'} = T_{ps} + \delta$ where $\delta > 0$. The same restriction as described in Definition 1 applies. If $T_{es} > T_L$, where T_L is a limit of a real-time deadline or acceptable delay, then the time for decryption of the encrypted strings is considered a hindrance to acceptable program operations.

Combined Effects of Control Flow Obfuscation and String Encryption

The total performance impact of obfuscation can be determined by combining Definitions 1 and 2.

Definition 3: Performance Effect of Obfuscation

Let $T' = T + \alpha + \delta$ show the effect of both control flow (α) and string decryption (δ) have on the original system performance. It is important to consider both effects on performance since it is important to not rely solely on one effect for the protection of a program. Three effects shown will have an effect on security as well as an impact on performance.

Test Results

Four preliminary tests were conducted to calculate the performance cost of various methods of obfuscation. The tests were conducted on a 3GHz Pentium 4 system running Fedora Core 6 system using Java 1.6 to compile the program, Zelix Klassmaster 5.0 trial version obfuscator, and GNU Compiler Collection 4.1.1 20070105 (Red Hat 4.1.1-51) to compile the driver. A C++ driver program was created to run the target Java class file as the 'root' user on the system for 50 times and calculate the average number of central processing unit clock cycles it took to execute the target class file. The results for the tests can be seen in Table 1.

The tests show that even for a simple example the control flow obfuscation and the string encryption has some impact on the performance of the system. None of the obfuscation methods improved the performance of the target application. The impact of obfuscation must be analyzed as a part of development in order to measure the impact on system performance and user experience. Further testing and refinement of these metrics will provide a means for program managers to evaluate the performance costs of the many different Java obfuscators on the market (and in the public domain.)

Table 1: *Obfuscation Tests*

Test	Time (cpu clock cycles)	Percentage difference
Unobfuscated Fibonacci	2.7069×10^8	0%
Fibonacci program with aggressive control flow obfuscation	2.71142×10^8	+0.17%
Fibonacci program with flow obfuscation string encryption	2.71478×10^8	+0.29%
Fibonacci program with aggressive control flow obfuscation and flow obfuscation string encryption[2]	2.71356×10^8	+0.24%

Conclusion

Obfuscation is a method (albeit imperfect) to protect the intellectual property rights of its creators. Obfuscation could also be thought of as a method of protection against reverse engineering by making it difficult for a hacker to obtain a high-level representation of Java source code in order to make changes. Obfuscation does not provide any sort of run-time protection like watermarking or calculated checksums at periodic locations.

Organizations need to consider strongly what information is being released when a piece of software is distributed. *It cannot be assumed that information hard-coded into a program will not be retrieved.* This is of considerable importance when evaluating software for release through foreign military sales or other coalition partner arrangements.

For those looking to secure their software, there are professional tools available

that make claims of high dependability. Many companies offer tools for both Java obfuscation as well as .NET obfuscation. Additional claims of these tools are that they reduce package size and increase efficiency. Evaluation of these claims is on our list of future work.

It is generally agreed that Java can be reverse engineered. Obfuscation only slows you down, but obfuscation also increases the costs of reverse engineering sufficiently to deter many economic motives for reverse engineering. Anyone who dismisses obfuscation has probably not tried to reverse engineer non-trivial programs. Reverse engineering of militarily sensitive software is not constrained by the same economics as commercial software.

Why is Java used in defense software? Reducing development costs is one reason. Often, after the software has been delivered, there are compelling reasons to make the software available under foreign military sales. It is then too late to observe that Java should not be used and translating millions of lines of code of Java into something else is not a feasible option. What do you do? Obfuscation certainly does not solve this problem, but it is an option that government program managers acquiring software-intensive systems should be aware of as well as the larger issue of programming language selection in terms of software requirements and design.◆

References

1. "LOCO: An Interactive Code (De)Obfuscation Tool." ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation, 2006.
2. "Cracking String Encryption in Java Obfuscated Bytecode." Subere 2006 <www.milw0rm.com/papers/117>.
3. "The Java Class File Disassembler." Java Sun <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javap.html>.
4. Miecznikowski, J., and L. Hendren. "Decompiling Java Using Staged Encapsulation." Proc. of the 8th Conference on Reverse Engineering, 2001.
5. Low, D. "Java Control Flow Obfuscation." Thesis. University of Auckland, 1998 <www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/>.

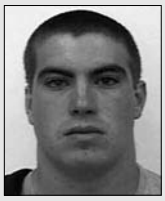
Notes

1. The Fibonacci numbers are the sequence of numbers $\{F_n\}_{n=1}^{\infty}$ defined by the *linear recurrence equation* $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. As a result of the definition, it is conventional to define $F_0 = 0$. (Wolfram Math Word <http://mathworld.wolfram.com/FibonacciNumber.html>).
2. The average time of aggressive control flow obfuscation and string encryption is most likely due to the fact that the control flow obfuscation has be optimized in some manner.

About the Authors

Stephen Torri is a doctoral candidate at Auburn University. He has a bachelor of science in accounting, finance, and computer science from Lancaster University in the United Kingdom and a master of science in computer science from Washington University in Saint Louis. Torri was an electronics technician in the U.S. Navy's Nuclear Power Program as a reactor operator aboard the USS Carl Vinson.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-7002
Fax: (334) 844-6329
E-mail: torrisa@auburn.edu

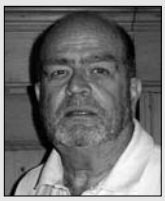


Derek Sanders is a graduate student studying Data Networks and Information Assurance with Auburn University. He is currently pursuing his masters degree in software engineering. Sanders' research interests include the medium access control layer for wireless communication, computer and network security, and a wide selection of issues related to securing wireless communications.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-7002
Fax: (334) 844-6329
E-mail: sandede@auburn.edu

John A. "Drew" Hamilton Jr., Ph.D., is an associate professor of computer science and software engineering at Auburn University and director of its information assurance laboratory. Prior to his retirement from the U.S. Army, he served as the first director of the Joint Forces Program Office and on the staff and faculty of the U.S. Military Academy, as well as chief of the Ada Joint Program Office. Hamilton has a bachelor's degree in journalism from Texas Tech University, masters degrees in systems management from the University of Southern California and in computer science from Vanderbilt University, as well as a doctorate in computer science from Texas A&M University.

Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, AL 36849
Phone: (334) 844-6360
Fax: (334) 844-6329
E-mail: hamilton@auburn.edu



Gordon Evans retired from the U.S. Army in 1992 as a Lieutenant Colonel. During his military service, he served in multiple field artillery, military intelligence, and overseas assignments. Since his retirement, Evans has worked as an on-site consultant to the Missile Defense Agency (MDA) where his areas of concentrations include systems engineering, command and control, modeling and simulations, international programs, and export control and technology transfers. He has been the lead MDA designer and investigator for its modeling and simulation vulnerability assessment program.

MDA
7100 Defense Pentagon
ATTN: MDA/BC
Washington, D.C. 20301-7100
Phone: (703) 697-4582
Fax: (703) 695-6133
E-mail: gordon.evans.ctr@mda.mil

Advancing Defect Containment to Quantitative Defect Management

Alison A. Frost and Michael J. Campo
Raytheon

The defect containment measure is traditionally used to provide insight into project success (or lack thereof) at capturing defects early in the project life cycle, i.e., the time when defect repair costs are at their minimum. Although the measure does provide insight into the effectiveness of early defect capture techniques (such as peer reviews), defect containment in its most common form (percentage of defects captured) is a lagging indicator as its ultimate value cannot be known until a project is complete. At that point, it is too late for a project to take corrective action. Using raw defect containment data and deriving Quantitative Defect Management (QDM) measures early in the development life cycle provides opportunities for a project to identify issues in defect capture before costs spiral out of control, schedule delays ensue, and another Death March begins [1].

Software quality issues have become a sad cliché in the software engineering industry. Versions 1.0 of commercial software products are notoriously defect-ridden. Furthermore, mission critical software has exhibited spectacular disasters, such as the loss of the Mars Climate Orbiter when English units were used in the coding of the ground software file used in trajectory models rather than the specified metrics units [2]. Ensuring software quality in mission critical systems is a primary cost driver in software development.

Several leading industry experts have analyzed defect injection rates during software development. Watts Humphrey found, " ... even experienced software engineers normally inject 100 or more defects per KSLOC [thousand lines of code] into their programs" [3]. Capers Jones gathered, "A series of studies found the defect density of software ranges from 49.5-94.6 errors per thousand lines of code" [4].

Compounding this situation, defects detected late in the development cycle cost many times more to repair than defects detected in the stage they were injected. For example, Watts Humphrey's research showed that the time it takes to fix a defect that escapes out-of-stage as shown in Table 1 [3].

Defect Containment Basics

Many companies employ a defect containment strategy in an attempt to reduce software costs and increase software quality. Programs and/or organizations may provide monthly resulting measures from this strategy as part of their team feedback or management reviews. Defect containment divides the engineering development cycle into separate stages and maps the stage in which a defect originated to the stage in which the defect was detected (see Table 2).

Defects may originate at any stage of the software development life cycle (although usually the greatest percentage of defects originates in the code and unit test stage). Defects detected in-stage are typically those defects detected during peer reviews or unit tests. Defects detected out-of-stage are those detected after the work product (e.g., design specification, or code) has been delivered to a downstream user (e.g., design released to development team or code released to software integration team). In-stage defects appear along the diagonal cells. (In Table 2, 2,421 defects originated and were detected during code and unit test.) Out-of-stage defects appear in the cells below the diagonal. (In Table 2, 1,525 defects originated in code and unit test, but were not detected until software integration.) These defect data provide insights to identify which processes cause the most defects and which processes allow defects to escape.

Defect containment is usually reported as percentages of defects captured in the stage in which they originated (see Table 3).

Using the data from Table 2, 48 percent of defects originating in design were detected (contained) in the design review process; 55 percent of defects originating in code were detected in the code review/unit test process; and the overall defect containment which equal the total number of defects caught in-stage/total defects for Table 2 was the following:

(1,515+1,555+2,421+37+1+10+0)/11,292 = 49 percent.

However, using defect containment to measure effectiveness as a percentage of in-stage capture is a lagging indicator. Until a project has gone through the later development stages, the ultimate number of defects injected is unknown. Furthermore, reporting superlative in-stage capture rates prior to qualification testing and system integration can be very misleading. The effectiveness of design and code peer reviews is unknown until it is too late for a project to take action. As such, traditional defect containment becomes a useful post-mortem tool, but does little to help a project when the project still has an opportunity to take corrective action.

Unfortunately, these two defect containment matrices (i.e. raw data count and percentage) are where the majority of engineers and managers conclude their defect data examination. However, by implementing a few derived measures from the defect containment base measures, one can employ proactive QDM.

Table 1: *Time to Fix Defect That Escapes Stage (in hours)*

Requirement	Design	Coding	Development Test	Acceptance Test	During Operation
1	3-6	10	15-40	30-70	40-1,000

QDM

QDM predicts the number of defects expected to be detected in each stage of software development, enabling proactive measures to be taken early in development. Why wait until system integration to discover that design and code peer reviews were ineffective? QDM allows a project to compare its defect detection rates against similar projects. These predictive and leading (as opposed to lagging) software measurements provide a mechanism to deter defect-driven cost and schedule overruns. This measure can be reported to a program and/or organization periodically (e.g. monthly) along with the defect con-

Stage Detected	Stage Originated							
	Requirements	Design	Code and Unit Test	SW Integration	SW Quality Test	System Integration and test	SW Maintenance	Total
Requirements	1,515							1,515
Design	1,181	1,555						2,736
Code and Unit Test	402	912	2,421					3,735
SW* Integration	200	420	1,525	37				2,182
SW Quality Test	191	223	370	7	1			792
System Integration and Test	89	114	114	5	0	10		332
SW Maintenance	0	0	0	0	0	0	0	0
Total	3,578	3,224	4,430	49	1	10	0	11,292

* SW = Software

Table 2: *Software Defect Containment Matrix*

tainment measures for team or management analysis and review.

Benefits of QDM are the following:

- Using predictive defect measures, a project knows in real time if it is meeting expected defect detection performance. For example, if a project is not finding the expected defects in design and code reviews, managers should investigate to determine if there is a reasonable cause or if corrective action is needed.
- Underperforming projects gain the ability to make corrective actions early rather than discovering problems at the end of the project.
- Overachieving projects provide the organization a chance to share best practices and lessons learned.
- Quantitatively understanding the capability of its peer review process offers an organization a chance to establish goals for defect capture and prevention, laying the groundwork for continuous improvement activities, and establishing Capability Maturity Model Integration (CMMI®) high maturity processes. (Please note: Although QDM may be a component of a CMMI high maturity process, by itself it may not qualify an organization to be rated CMMI Maturity Level 4 or 5.)

There are five key factors to take into account when applying QDM. To be effective, an organization must do the following:

1. Utilize *consistent definitions* for terms such as *defect*, *size unit* (e.g. source lines of code [SLOC]), and *life-cycle stages*.
2. Automate data collection and reporting to record and track defect data. Many change request tools exist that facilitate the recording and retrieval of in-stage and out-of-stage defect data, as well as automating the creation of derived measurement charts for projects. *Exploitation of automation allows projects to focus on data analysis rather than collection.*
3. Use *past data* to analyze current performance and predict future performance. Doing so allows one to create and maintain control limits based on performance capability. One can manage based on quantitative analysis.
4. Involve and train *all levels of personnel*. Besides improving data integrity, practitioners' perspectives and analyses are often found to be the most valuable. Ownership of organizational goals becomes shared by all levels of personnel.
5. Use QDM to improve project and organizational performance, *not to target individuals*. This is true of any measure.

QDM aligns with many industry initiatives. For example, QDM supports CMMI Level 4 and Level 5 as well as Six Sigma philosophies [5].

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Table 3: *Software Defect Containment Percentage Matrix*

Stage Detected	Stage Originated						
	Requirements	Design	Code and Unit Test	SW Integration	SW Quality Test	System Integration and test	SW Maintenance
Requirements	42%						
Design	33%	48%					
Code and Unit Test	11%	28%	55%				
SW Integration	6%	13%	34%	76%			
SW Quality Test	5%	7%	8%	14%	100%		
System Integration and Test	2%	4%	3%	10%	0%	100%	
SW Maintenance	0%	0%	0%	0%	0%	0%	0%

In order to use defect containment in this predictive manner, organizations must do the following: 1) establish a baseline by using defect containment data from previously completed projects, 2) normalize the defect data found in each stage by size (e.g. SLOC), and 3) apply statistical techniques to set limits of expected defect detection performance.

Three QDM Measures

Three measures derived from the defect containment matrix that offer immediate proactive insight into defect data are the following: 1) Cumulative Defects Originated in Design Detected by Stage; 2) Cumulative Defects Originated in Code and Unit Test Detected by Stage; and 3) Defect Detection Distribution by Stage.

The following steps will cover required base measures, establishment of control limits/boundaries, and graphical representation of data. (More measures can be derived from defect containment to offer proactive insight, but this sample is a good start for a wide range of software engineers.)

In order to create these three QDM measures, the following base and derived measures are required:

- Number of defects by stage of origin (leveraged directly from the defect containment matrix [see Table 2]).
- Number of defects by stage of discovery (leveraged directly from the defect containment matrix [see Table 2]).
- A size count, such as SLOC or function points.
- Normalize the defect count by the size count (e.g. x defects per KSLOC).

Next, use comparison techniques on historical data to establish the range of expected defect detection, i.e., control limits. These data must come from independent observations of the same process (e.g. separate design reviews.) The data from each life-cycle stage is compared to data from its own stage. In cases where a defect in an earlier stage causes a defect in a later stage, the defect counts as a single defect in the stage it was originally introduced.

Control limits can be derived by calculating 3σ limits based on existing data. In this manner, defect data will fall between these (3σ) limits 99.7 percent of the time. Using 3-sigma limits avoids the need to make assumptions about the distribution of the underlying natural variation. As noted by Florac and Carleton in the following note:

> ... experience over many years of control charting has shown 3-sigma limits to be economical in the sense that they are adequately sensitive to unusual variations while leading to very few (costly) false alarms – regardless of the underlying distribution. [6]

Note: To calculate the control charts in these examples, the u-chart formulas were used.

For Cumulative Defects Originated in Design Detected by Stage (Figure 1) and for Cumulative Defects Originated in Code and Unit Test Detected by Stage (Figure 2), 3-sigma control limits are established using the following u-chart formulas:

$$UCL = \bar{u} + 3\sqrt{\frac{\bar{u}}{n_j}},$$

$$LCL = MAX\left[0, \bar{u} - 3\sqrt{\frac{\bar{u}}{n_j}}\right],$$

where ubar is the mean for each subgroup and n_j is the sample size. An example follows in Table 4. Note: In this example, KSLOCs were the size units of the design artifacts.

For Defect Detection Distribution by Stage (Figure 3), utilize a set of greater/less than boundaries. The number of defects detected in the software requirements stage should be less than the number found in the design stage. The number of defects detected should continue to increase through the code and unit test stage. After the code and unit test stage, the defects detected in each stage should decrease through the remaining stages with software maintenance stage detecting the least amount of defects.

Finally, plot the data.

For the Cumulative Defects Originated in Design Detected by Stage, create a chart where the x-axis is the life-cycle stage and the y-axis is the number of detected design-originated defects normalized by size unit. Plot the total normalized number of design defects found in-stage, followed by the total cumulative numbers of design defects detected in each subsequent life-cycle stage (code and unit test, software integration, software qualification test, system integration, and software maintenance). Pending analysis preference, data points may or may not be connected as a line on the chart; in the examples that follow, they are connected (see Figure 1).

For the Cumulative Defects Originated in Code and Unit Test Detected by Stage, create the charts similar to the process reviewed for Cumulative Defects Originated in Design Detected by Stage (see Figure 2).

For the Defect Detection Distribution by Stage, plot a chart where the x-axis is the software life-cycle stage and the y-axis is the normalized number of defects detected in each stage, regardless of the stage in which they were introduced (see Figure 3 for a display of the measure).

For Defect Detection Distribution by Stage, defect detection distribution ideally will mirror the defect injection distribution (thereby capturing defects as close as possible to when they were injected). It is known that the defect injection rate maps to the Rayleigh distribution curve as shown in Figure 4 [7]. (Statistically, the Rayleigh distribution is a Weibull Distribution with a value of two.) Therefore, it can be used to track the pattern of defect removal during the software life cycle.

Table 4: *Cumulative Defects Originated in Design Detected by Stage Control Limits*

Lifecycle Stage Where Design Defects Detected	Design Defects/Actual KSLOC	
	Minimum	Maximum
Design	3.7	9.9
Code and Unit Test	4.4	10.6
SW Integration	4.7	10.9
SW Quality Test	4.9	11.1
System Integration and Test	5.4	11.6
SW Maintenance	5.4	11.6

Analysis Results

Analysis of the QDM measures indicates the best course of action for the project and organization. Further, it is important to compare the QDM measures with other measures that the program or orga-

nization maintain in order to obtain a more complete understanding. Ultimately, the QDM measures provide indicators for further investigation. Opportunities to improve performance will vary among projects and organizations, as shown in the following examples:

- If a current project falls above the upper control limit, a course of action may be to perform causal analysis to understand the reason for the behavior. Possible actions include investigating means to reduce defects injected, adjusting control limits, and identifying best practices for defect detection to be considered for organizational deployment.
- If a current project falls below the lower control limit, a goal may be to get the current project to be as effective (e.g. during peer review) as the past projects; this would be demonstrated by moving the project within the control limits over time. In this case, the project may aggressively work to improve design and code peer reviews.
- Different opportunities exist if the project data falls within the control limits. Options include deploying defect prevention measures that drive the data toward the lower control limit of the charts illustrated in Figures 1 and 2. Alternatively, one may choose to gather a large enough sample to tighten the existing control limits and decrease projected variability.
- When looking at the Defect Detection Distribution by Stage measure, if the project has more defects detected in the design stage than the code stage (the defect detection efforts during code and unit testing may have not been effective), the project may not be ready to begin the software integration effort.

Establishing control limits on defect detection provides an organization the ability to predict the number of defects that will be inserted into project work products, based on work product size and the use of a standard organizational software development process. Predicting defects inserted within a statistically derived range may be used to determine readiness to move from one development stage to the next, and to predict future rework costs.

Further, utilizing organization data or industry standards on *hours to correct defects by stage*, return on investment can be calculated. Identifying peer review process or training issues can provide substantial savings for minimal investment.

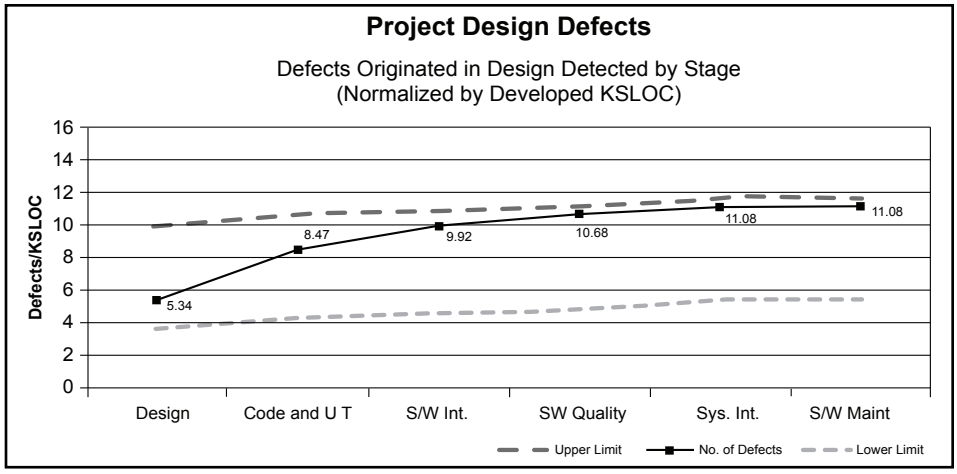


Figure 1: *Cumulative Defects Originated in Design Detected by Stage*

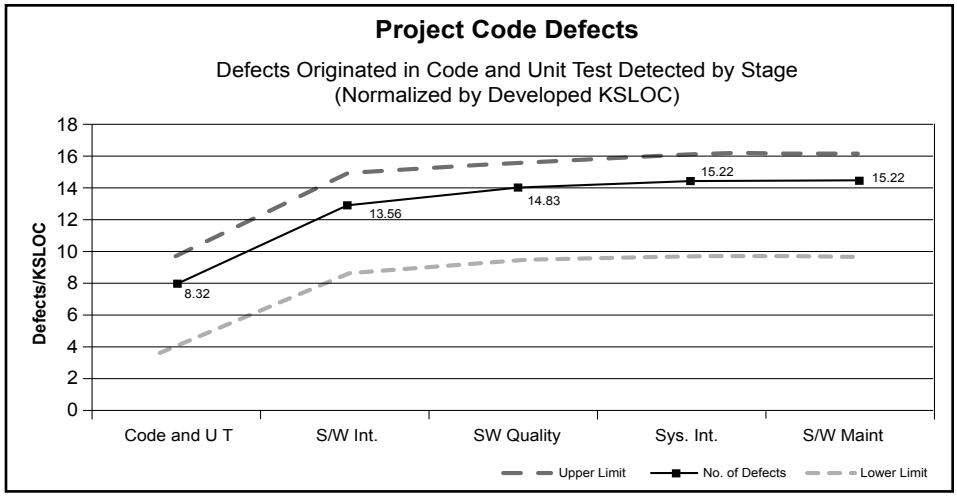


Figure 2: *Cumulative Defects Originated in Code and Unit Test Detected by Stage Chart*

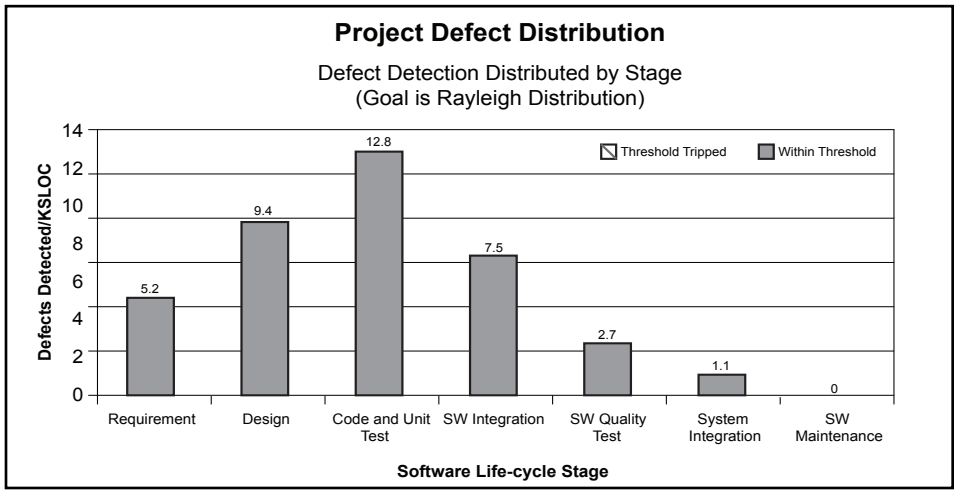


Figure 3: *Defect Detection Distribution by Stage*

Some examples of actual process improvements that resulted from the use of the QDM (implemented at Raytheon organizations) include the following:

- Design and code peer review standards were improved, with recommendations of:
  - Design peer review preparation rate of less than 250 SLOC per hour per reviewer.
  - Code peer review preparation rate

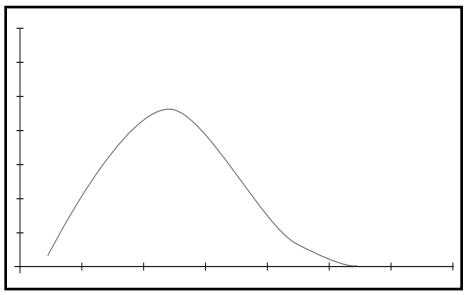


Figure 4: *Rayleigh Distribution*

of less than 200 SLOC per hour per reviewer.
  - Peer reviews meetings should not last longer than two hours [8].
- Peer reviews are postponed when participation is inadequate.
- Project meetings are held to provide feedback on QDM measures, address training, and investigate questions of data integrity.
- Software measurement tools were updated to improve automation of data collection and support analysis.

The improved peer review process, data entry and analysis, and measurement automation, were direct results of the QDM efforts.

Conclusion

QDM takes defect containment to a new level – from a reactive, lagging indicator to a proactive, predictive indicator of software quality. Samplings of derived defect measures with steps on how to create them were offered. QDM analyses provide an array of opportunities for process improvements that increase quality and reduce costs at both project and organizational levels.◆

References

1. Yourdon, E. Death March. 2nd ed. Prentice Hall, 2003.
2. Leveson, N. The Role of Software in Spacecraft Accidents. Massachusetts Institute of Technology, 2004.
3. Humphrey, W. "A Personal Commitment to Software Quality." Pittsburgh, PA: The Software Engineering Institute (SEI) <www.sei.cmu.edu>.
4. Jones, T.C. Programming Productivity. New York: McGraw-Hill, 1972.
5. SEI. Capability Maturity Model Integration. Version 1.2. Carnegie Mellon, SEI, 2006.
6. Florac, William A., and Anita D. Carleton. Measuring the Software Process. Addison Wesley, 1999.
7. Kan, Stephen H. Metrics and Models in Software Quality Engineering. Addison-Wesley Publishing Company, 1995.
8 Frost, A.A. "Design and Code Inspection Metrics." International Conference on Applications of Software Measurement, San Jose, CA, 1999.

About the Authors

Alison A. Frost is a national process engineer at Raytheon. While in Massachusetts, she was a leader in the achievement of CMM Level 4 for a 550-person Massachusetts/Alabama laboratory. Frost managed an engineering process group (EPG) metrics team comprised of statisticians, tools personnel, and systems/software engineers. In California, she was the DD(X) software measurement lead where she spearheaded the deployment of a measurement repository to more than 30 sites and companies. Currently, Frost is a Network Centric Systems Fullerton EPG member; her recent highlight was contributing the organization's CMMI software engineering/software/hardware Level 5 rating.

Raytheon
Fullerton, CA 92834
Phone: (978) 590-5905
Fax: (801) 340-7108
E-mail: frostfrost @earthlink.net

Michael J. Campo is a principal engineering fellow at Raytheon where he currently leads the Raytheon integrated defense systems (IDS) EPG and is a member of the Raytheon corporate CMMI expert team. Campo is a metrics leader at Raytheon. He is also an SEI-authorized CMMI Lead Appraiser and CMMI Instructor. He led Raytheon IDS to a CMMI Level 3 software engineering/software rating in 2003, and CMMI software engineering/software Level 4 and hardware Level 3 rating in 2005.

Raytheon
Tewksbury, MA 01876
Phone: (978) 858-5939
Fax: (978) 858-4505
E-mail: michael_j_campo @raytheon.com

CROSSTALK
The Journal of Defense Software Engineering

ARTICLE INDEX VOLUME 19

TOPIC	ARTICLE TITLE	AUTHOR(S)	ISSUE	PAGE
Acquisition	The Acquisition of Joint Programs	Dr. Mary Maureen Brown, Robert M. Flowe, Sean Patrick Hamel	5	20
	Controlling Software Acquisition Costs With Function Points and Estimation Tools	Ian Brown	5	9
	Defense Acquisition Performance Assessment – The Life-Cycle Perspective of Selected Recommendations	Dr. Peter Hantos	5	25
	Software Acquisition in the Army	Elizabeth Starrett	5	4
	Software Assurance: Five Essential Considerations for Acquisition Officials	Mary Linda Polydys, Stan Wisseman	5	14
Agile Development	Collaboration Skills for Agile Teams	Esther Derby	4	8
	Toward Agile Systems Engineering Processes	Dr. Richard Turner	4	11
	What Engineering Has in Common With Manufacturing and Why It Matters	Dr. Alistair Cockburn	4	4
Change Management	Controlling Organizational Change: Beyond the Nightmare	Deb Jacobs	8	12
CMMI	CMMI V1.2: What Has Changed and Why	Mike Phillips	2	4
	Connecting Software Industry Standards and Best Practices: Lean Six Sigma and CMMI	Gary A. Gack, Karl D. Williams	2	Online
	Profiles of Level 5 CMMI Organizations	Donald J. Reifer	1	24
COTS	Added Sources of Costs in Maintaining COTS-Intensive Systems	Dr. Betsy Clark, Dr. Brad Clark	6	4
	Applying COTS Java Benefits to Mission-Critical Real-Time Software	Dr. Kelvin Nilsen	6	19
	GL Studio Brings Realism to Aircraft Cockpit Simulator Displays	Kim Stults	6	16
	Issues to Consider Before Acquiring COTS	Dr. David A. Cook	6	9
	Lean AISF: Applying COTS to System Integration Facilities	Harold Lowery	6	13
Miscellaneous	Good News From Iraq	CAPT Steven J. Lucks (Ret.)	8	4
	"OO-OO-OO!" The Sound of a Broken OODA Loop	Dr. David G. Ullman	4	22
Net-Centricity	Challenges of Internet Development in Vietnam: A General Perspective	Duy Le, Dr. Rayford B. Vaughn, Dr. Yoginder S. Dandass	1	16
	Communicating on the Move: Mobile Ad-Hoc Networks	Robert F. Dillingham, Dean Nathans	7	22
	Enabling Technologies for Net-Centricity – Information on Demand	The Honorable John J. Grimes	7	4
	For Net-Centric Operations, the Future Is Federated	John Michelsen	9	24
	Getting to GIG: Enterprise-Wide Systems Engineering	Defense Information Systems Agency	7	9
	Making Information Visible, Accessible, and Understandable: Meta-Data and Registries	Clay Robinson	7	17
	Making It Work – The Net-Centric Global Information Grid NetOps Strategy	Thomas Lam	7	11
	Managing the Air Waves: Dynamic Spectrum Access and the Transformation of DoD Spectrum Management	Thomas J. Taylor	7	19
	Net-Centric Conversations: The Enterprise Unit of Work	Harvey Reed, COL Fred Stein (Ret.)	8	18
	Net-Centric Operations: Defense and Transportation Synergy	COL Kenneth L. Alford, Ph.D., Steven R. Ditmeyer	1	20
	Providing the Tools for Information Sharing: Net-Centric Enterprise Services	Ann H. Kim, Carol Macha	7	10
	Reconfiguring to Meet Demands: Software-Defined Radio	Dean Nathans, Dr. Donald R. Stephens	7	24
	Securing the Global Information Grid – The Way Ahead for Information Assurance	Richard Aldrich, David Zaharchek	7	13
	Sharing Information Today: Maritime Domain Awareness	Michael Todd	7	28
	Sharing Information Today: Net-Centric Operations in Stability, Reconstruction, and Disaster Response	Dr. Linton Wells, II	7	7
	Spiraling Information Demands – The Way Ahead With IPv6	Kristopher L. Strance	7	15
	Trusting the Team: Identity Protection and Management	Defense-Wide Information Assurance Program	7	20
	A Unified Service Description for the Global Information Grid	Dr. Yun-Tung Lau	8	23
	Using Switched Fabrics and Data Distribution Service to Develop High Performance Distributed Data-Critical Systems	Dr. Rajive Joshi	4	26

CONTINUED ON NEXT PAGE

From Previous Page

TOPIC	ARTICLE TITLE	AUTHOR(S)	ISSUE	PAGE
Process Improvement	Applying International Software Engineering Standards In Very Small Enterprises	Claude Y. Laporte, Alain April, Alain Renault	2	29
	CMMI Level 2 Within Six Months? No Way!	George Jackelen	2	13
	Future Directions in Process Improvement	Watts S. Humphrey, James W. Over, Dr. Michael D. Konrad, William C. Peterson	2	17
	The ImprovAbility Model	Dr. Jan Pries-Heje, Mads Christiansen, Jørn Johansen, Morten Korsaa	2	23
	Lessons Learned in Using Agile Methods for Process Improvement	Nelson Perez, Ernest Ambrose	8	7
	Measure Twice and Cut Once	Rushby Craig	2	8
	Why Should I Use the People CMM?	Margaret Kulpa	11	19
Project Management	Earned Value Management: Are Expectations Too High?	LTC Nannette Patton, Allan Schechet	1	10
	Tools for Decision Analysis and Resolution	Dr. Richard D. Stutzke	11	23
Quality	Advancing Defect Containment to Quantitative Defect Management	Alison A. Frost, Michael J. Campo	12	24
	Beyond Defect Removal: Latent Defect Estimation With Capture-Recapture Method	Joe Schofield	8	27
	The Relative Cost of Interchanging, Adding, or Dropping Quality Practices	Bob McCann	6	25
Service-Oriented Architectures	Applying a Service-Oriented Architecture to Operational Flight Program Development	Mitch Chan	9	20
	Common Misconceptions About Service-Oriented Architecture	Grace A. Lewis, Edwin Morris, Dr. Dennis B. Smith, Soumya Simanta, Lutz Wrage	11	27
	Defining Services Using the Warfighter's Language	Michael S. Russell	9	14
	Four Pillars of Service-Oriented Architecture	Grace A. Lewis, Dr. Dennis B. Smith	9	10
	Reference Metrics for Service-Oriented Architectures	Dr. Yun-Tung Lau	12	15
	SOA Security Reference Model	Nataraj Nagaratnam, Anthony Nadalin, Janet Mostow, Sridhar Muppidi	9	Online
Software Security	Baking in Security During the Systems Development Life Cycle	Kwok H. Cheng	3	22
	Being Explicit About Security Weaknesses	Robert A. Martin	3	4
	Cross-Domain Information Sharing in a Tactical Environment	Mel Crocker	3	26
	High-Leverage Techniques for Software Security	Idongesit Mkpong-Ruffin, Dr. David A. Umphress	3	18
	How a Variety of Information Assurance Methods Delivers Software Security in the United Kingdom	Kevin Sloan, Mike Ormerod	3	13
	A Primer on Java Obfuscation	Stephen Torri, Derek Sanders, Dr. Drew Hamilton, Gordon Evans	12	19
	Secure Coding Standards	James W. Moore, Robert C. Seacord	3	9
	The Security of Web Services as Software	Karen Mercedes Goertzel	9	4
	Software as an Exploitable Source of Intelligence	Dr. David A. Umphress	6	29
Software Sustainment	Geriatric Issues of Aging Software	Capers Jones	12	4
	Performance-based Software Sustainment for the F-35 Lightening II	Lloyd Huff, George Novak	12	9
Systems Engineering	ConOps: The Cryptex to Operational System Mission Success	Alan C. Jost	10	13
	A Framework for Evolving System of Systems Engineering	Dr. Ricardo Valerdi, Dr. Adam M. Ross, Dr. Donna H. Rhodes	10	28
	Issues Using DoDAF to Engineer Fault-Tolerant Systems of Systems	Dr. Ronald J. Leach	10	22
	Software System Engineering: A Tutorial	Dr. Richard Hall Thayer	10	17
	Systems Engineering for the Global Information Grid: An Approach At the Enterprise Level	Patrick M. Kern	10	10
	Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering	Dr. Barry Boehm, Jo Ann Lane	10	4
	Where Hardware and Software Meet: The Basics	Mike McNair	1	6
Team Software Process	CMMI Level 5 and the Team Software Process	David R. Webb, Dr. Gene Miluk, Jim Van Buren	4	16
Working as a Team	The Gauge That Pays: Project Navigation and Team Building	Kasey Thompson, Tim Border	11	14
	Shaping Motivation and Emotion in Technology Terms	Jennifer Tucker, Hile Rutledge	11	10
	Wisdom for Building the Project Manager/Project Sponsor Relationship: Partnership for Project Success	LTC Nanette Patton, Allan Shechet	11	4

I Want My BACKTALK Back

I just received notice from CROSSTALK's managing editor requiring a BACKTALK article for the December issue. Now!

I apologize in advance for the lack of preparation. A previous message indicated that BACKTALK was covered, and Dr. Cook and I could take a couple of issues off. In reviewing the message, the two issues were January and February, and I'm on the line for December.

No problem. I've had short deadlines before. I work well under pressure. Let the creative juices flow from the cortex to the fingers through the keyboard to the screen past the editor to the page.

Wait, what does the last sentence of the message say?

"It needs to be short, as we are printing the article index – so about half of what you typically write."

Are you kidding me? They are cutting BACKTALK short for an article index? Do they really think CROSSTALK readers wait with baited breath to meticulously browse the article index? Do they realize this is the 21st century and indexes belong on the Web?

It needs to be short? Meetings need to be short. Commercials need to be short. Queues need to be short. Computer start-up times need to be short, not BACKTALK. It's already concise and to the point.

Would they treat Gustave Eiffel this way? "Hey Gustave, beautiful tower, can you make that half as high? We don't want to distract the tourists ... merci."

They obviously got to da Vinci. "Hey Leonardo, here's an offer you can't refuse. I want Mona's beaming smile diluted to a half a grin ... grazie."

Half of what I typically write? Would you ask Dennis Miller for half a rant; Krispy Kreme for half a doughnut; Britney for half a rehab – okay, I'll give you that; Lance for half an effort; the Wizard for half a brain, heart, or courage? Would you ask Al Gore for half a carbon footprint? Wait, we did and he won't. Hey, there's an idea, if I buy an arti*cle index offset* can I take more space for BACKTALK?

I understand half the calories, half the wait, or a half-off sale, but half a BACKTALK makes no sense. What are they thinking? Can you imagine the CROSSTALK boardroom conversation? "We need space for the annual article index. Where can we find space?"

"What about BACKTALK?"

Sting faintly singing: *I want my* BACKTALK *back*.

Synthesizer, drums, guitar. Dire Straits sings:

Look at them yo-yo's, that's the way to do it;
You write BACKTALK for all to see.
That ain't workin', that's the way you do it,
Article for nothing and your kicks for free.

Now that ain't workin',
That's the way you do it,
Let me tell ya – them guys ain't dumb
Maybe get a blister on their little finger
Maybe get a blister on their bum

We gotta install article indexes,
Custom issue deliveries
We gotta move that impersonator,
We gotta move that Ph.D.

Sting boldly singing: *I want my, I want my, I want my* BACKTALK *back*.

Let's be honest: We all know that nine out of 10 CROSSTALK readers turn to BACKTALK straight away for wit, indulgence, and inspiration. While the Publisher's Note introduces the issue, BACKTALK sets the tone – warming up a reader's mind in preparation for the technical feast inside. If they think they can get away with a half-baked BACKTALK ...

Knock, knock, knock.

Pardon me a moment to get the door.

Yes? Excuse me, are you arresting me? What have I done? What have I done? What have I done! You're arresting me? Whoa, whoa, whoa, get off me. Get off me! Help! Why are they arresting me? What did I do? Get off me! Get off me! I didn't do anything!

...

We now return to your regularly scheduled (half) BACKTALK (soft, soothing elevator Muzak).

— Gary A. Petersen
Arrowpoint Solutions, Inc.
gpetersen@arrowpoint.us

MONTHLY COLUMNS

ISSUE		COLUMN TITLE	AUTHOR
Issue 1: January Publisher's Choice	Sponsor: Publisher: BackTalk:	Unit Compliance Inspection: What Did We Learn? Choose Your Favorite One If By LAN, Two If By C	Diane E. Suchan Elizabeth Starrett Dr. David A. Cook
Issue 2: February CMMI	Sponsor: BackTalk:	Axiomatic Improvement Hippocrates and the Oath	Randy B. Hill Gary A. Petersen
Issue 3: March Software Security	Publisher: BackTalk:	Collaborating for Secure Software Project Management Using Random Events	Elizabeth Starrett Dr. David A. Cook
Issue 4: April Agile Development	Sponsor: BackTalk:	"Lead, Follow, or Get Out of the Way" (Un) Due Diligence	Kevin Stamey Dr. David A. Cook
Issue 5: May Software Acquisition	Sponsor: BackTalk:	Being a Smart Buyer Who Are Those Guys?	Tony Guido Gary A. Petersen
Issue 6: June COTS Integration	Sponsor: BackTalk:	Navigating the COTS Sea COTS: Commercial Off-The-Shelf or Custom Off-The-Shelf?	Diane E. Suchan Wiley F. Livingston, Jr., P.E
Issue 7: July Enabling Technologies for Net-Centricity	Sponsor: BackTalk:	Delivering the Power of Information Net-Centric Virtuosity	General James E. Cartwright Gary A. Petersen
Issue 8: August Stories of Change	Sponsor: BackTalk:	The Right Way to Change Common Threads in Life	Norman R. LeClair Glenn Booker
Issue 9: September Service-Oriented Architecture	Publisher: BackTalk:	SOA Provides Opportunities and Challenges Evolution in Action – Building Up to a Service-Oriented Architecture	Elizabeth Starrett Dr. David A. Cook
Issue 10: October Systems Engineering	Sponsor: BackTalk:	Revitalization of Systems Engineering Within the Department of Defense and the Expanding Role of Software Softwareitaville	Dr. John W. Fischer Gary A. Petersen
Issue 11: November Working as a Team	Sponsor: BackTalk:	Working as a Team SSMART Team Management	Kevin Stamey Dr. David A. Cook
Issue 12: December Software Sustainment	Publisher: BackTalk:	Software Sustainment or Maintenance? I Want My BACKTALK Back	Elizabeth Starrett Gary A. Petersen

CrossTalk / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CrossTalk is co-sponsored by the following organizations:

NAVAIR

Homeland Security